Modeling Healing Behaviors of Cyber-Physical Systems with Uncertainty to Support Automated Testing

Tao Ma[1], Shaukat Ali[1], Tao Yue[1, 2]
[1] Simula Research Laboratory, [2] University of Oslo
Oslo, Norway
{taoma, shaukat, tao}@simula.no

Abstract

Self-Healing Cyber-Physical Systems (ShCPSs) are capable of detecting and recovering from faults themselves during their operations, which are commonly referred to as self-healing behaviors. Testing such behaviors is challenging due to their self-adaptive nature and being often exposed to environment uncertainty. As a first step towards automated testing of ShCPSs in the presence of uncertainty, we propose a modeling framework called Modeling Self-Healing Behavior with Uncertainty (*MoSH*), which supports creating executable test ready models (ETRMs) of a ShCPS together with uncertainty in its environment. *MoSH* consists of four UML profiles, derived based on a conceptual model for ShCPSs (which is named as Conceptual Model for ShCPSs and Uncertainty (*CMSU*)), and a modeling methodology of applying these profiles to create ETRMs. In addition, we developed a test model execution framework called *TM-Executor* to support the execution of ETRMs to eventually enable testing of a ShCPS in the presence of various types of environment uncertainties. *TM-Executor* extends Moka—a standard-based UML model executor with new execution semantics specifically designed for test execution. In addition, *TM-Executor* integrates several tools including EsOCL for test data generation, DresdenOCL for test oracle checking, and the Functional Mockup Interface module for integrating simulators/emulators in order to introduce uncertainties in the environment of a ShCPS.

We assess *MoSH* and *TM-Executor* from three perspectives. First, we validated the completeness and correctness of *CMSU* with nine ShCPS case studies. Evaluation results show that *CMSU* is complete and correct. Second, we validated *MoSH* with three ShCPS case studies and results show that we were successful in creating ETRMs for the three case studies at the expense of additional 16% of modeling effort. Third, we evaluated the feasibility of *TM-Executor* with a random testing strategy implemented by testing a real-world case study as a proof of concept. We assessed its performance in terms of performing various testing activities (e.g., test data generation) and observed that time required to perform such activities was very small, i.e., in the order of milliseconds. Furthermore, we found one fault in a self-healing behavior of the real-world case study during the testing, which was only revealed in the presence of environment uncertainties.

Keywords: Cyber-Physical Systems, Self-healing, Uncertainty, Model Execution, Model-Based Testing

1. Introduction

Cyber-Physical Systems (CPSs) are increasingly becoming autonomous to deal with dynamic environment situations [1]. One type of autonomous behaviors of CPSs is *Self-Healing Behaviors*, i.e., *the ability of a CPS to detect and recover from a fault during its operation by itself* [2]. Given the fact that CPSs, in general, face uncertainty in their behaviors due to the unpredictable environment conditions [3], self-healing behaviors are also prone to such uncertainty. Thus, ensuring the correctness of self-healing behaviors in the presence of environment uncertainty is important for their reliable operation and stresses the development of efficient testing approaches. In the rest of the paper, we refer to CPSs with self-healing behaviors as Self-Healing Cyber-Physical Systems (ShCPSs).

1.1 Overall Objective and Challenges

Our overall objective is *to develop systematic and automated testing techniques to test ShCPSs in the face of environment uncertainty*. However, testing such behaviors in the face of environment uncertainty is challenging due to several reasons: 1) Self-healing behaviors being autonomous are naturally difficult to predict at the design time, and 2) Unpredictable environment makes the self-healing behaviors behave in an even more unexpected manner. Based on these two general challenges, to address our overall objective, we raise the following three testing challenges: 1) T1: How to capture expected self-healing behaviors together with environment uncertainties? 2) T2: How to generate the best set of test data and simulate uncertainties in the best manner, based on captured expected behaviors and uncertainties? 3) T3: How to exercise expected self-healing behaviors with captured uncertainties to find faults cost-effectively?

1.2 Overall Approach

An overview of our approach is presented on the right side of Figure 1 and the left side of the figure presents a typical architecture of Model-Based Testing (MBT) approaches. In this section, we aim to differentiate our approach with the typical MBT approach for the purpose of clearly defining the scope and highlighting key contributions (Section 1.3). Note that both of the two architectures presented in Figure 1 aim for achieving the same objective and addressing the same challenges presented in Section 1.1.

As a type of model-based approaches, MBT approaches rely on models for handling complexity via abstraction [4]. The expected behavior of a ShCPS under Test (SUT) together with environment uncertainties are modeled as a *Test Ready Model* (TRM). A *Test Case Generator* then uses a *Test Strategy* to generate test cases, which are executed on the SUT by a

Test Case Executor with the aim of detecting faults that can only be found in the presence of uncertainties.

An alternative approach, which we opt for, is coined as Executable Model-Based Testing (EMBT) and is shown on the right side of Figure 1. The process of creation of TRMs is similar to MBT except that TRMs in EMBT are enriched with code execution semantics such that these models are executable and thus are called as Executable Test Ready Models (ETRMs). As opposed to MBT, which typically applies a static *Test Strategy* to generate test cases using a *Test Case Generator*, EMBT uses a *Dynamic & Adaptive Test Strategy* to guide a *Test Model Executor* to execute an ETRM. The execution of the ETRM in turns drives the execution of SUT. Based on the SUT's actual state, reflected by the ETRM, the *Test Model Executor* uses the test strategy to dynamically select optimal stimulus (in terms of finding a fault) to be sent to the ETRM. Given that environment uncertainties lead to uncertain behaviors (including self-healing behaviors) of SUT, to effectively find faults, it is necessary to dynamically adapt the execution of the ETRM based on the feedback received from SUT during test execution. Since a ShCPS interacts with its physical environment using sensors and actuators, for the purpose of testing, we simulate environment uncertainties by introducing uncertainties at interfaces of sensors and actuators using dedicated emulators/simulators. Doing so is required for both MBT and EMBT.

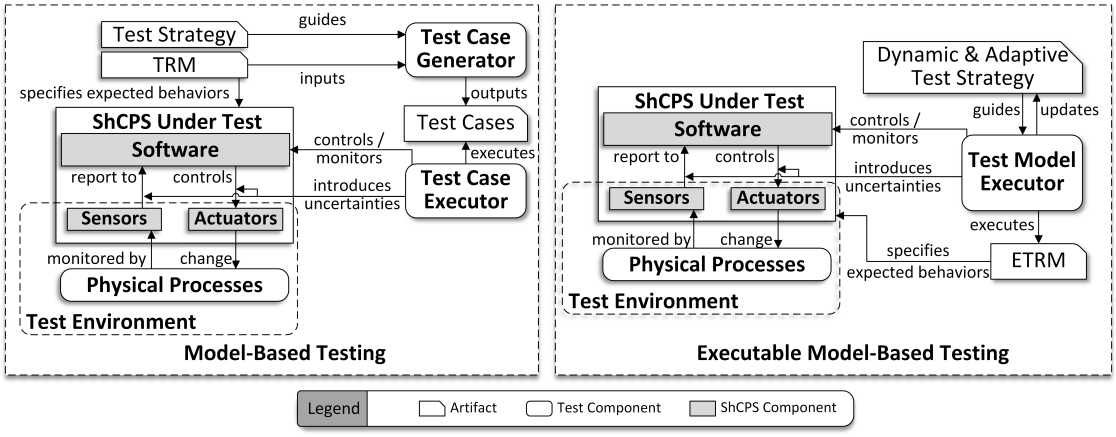

Figure 1 EMBT Comparing with MBT

1.3 Scope and Contributions

In Table 1, the 1st column presents the three testing challenges (Section 1.1); the 2nd column shows the mapping of the testing challenges with the key components of our EMBT solution, and the 3rd column shows the contributions of the paper and refers to the corresponding sections where details are discussed. One key summary is that the key contributions of this paper are the design, development and evaluation of *MoSH* and *TM-Executor*, for addressing *T1* and *T3*.

Notably, in the literature, there exist works for defining self-healing concepts [2, 5], based on an extensive study of which we derived *CMSU*. However, *CMSU* focuses on supporting

EMBT. Researchers have also spent effort on defining uncertainty concepts in the context of self-adaptive systems [6], CPSs [3], and a general context [7]. However, none of these works provide an end-to-end modeling solution to support testing of self-healing behaviors of CPSs in the presence of environment uncertainties, which is targeted in this paper.

Table 1 Testing Challenges, our EMBT Solution, and Corresponding Sections

Testing Challenges	Our EMBT Solution	Contributions and Corresponding Sections
T1: How to capture expected self-healing behaviors together with uncertainties in the environment?	ETRM Modeling Framework	*Modeling Self-Healing Behavior with Uncertainty (MoSH) Framework*—an ETRM modeling framework including 1) *Conceptual Model for ShCPSs and Uncertainty (CMSU)* (Section 3), 2) *MoSH Profiles*: the implementation of *CMSU* and 3) *MoSH Modeling Methodology* to develop ETRMs capturing expected behaviors of SUT and environment uncertainties (Section 4)
T2: How to generate the best set of test data and simulate uncertainties in the best manner, based on a the captured expected behaviors and uncertainties?	Test Strategies, Uncertainty Generation	This is not in the scope of this paper. However, as a proof of concept, we implemented a random strategy to select paths in an ETRM to execute, applied EsOCL [8]—a search-based test data generator to generate test data, and simulated uncertainties based on their universes, notions, and measures.
T3: How to exercise the expected self-healing behavior with captured uncertainty with the aim of finding faults cost-effectively?	Test Model Execution Framework	*TM-Executor*—a test model executor framework built on the model execution platform Moka [9], which implements the fUML standard [10]. *TM-Executor* provides a facility to perform adaptive testing and generate uncertainties for the testing of self-healing behaviors under uncertainties. *TM-Executor* integrates EsOCL [8] for test data generation, DresdenOCL [12] for test oracle (OCL constraints) checking, and a Functional Mockup Interface (FMI) [11] module for incorporating simulators/emulators in order to introduce uncertainties in the SUT's test environment. Details are presented in Section 5.

1.4 Evaluation

As discussed in Section 1.3 and summarized in Table 1, the contributions of this paper are *MoSH* and *TM-Executor*. Therefore, we focus on the evaluation of these two components. For *MoSH*, first, we evaluated *CMSU* (the conceptual model from which *MoSH* was derived) in terms of its completeness and correctness with nine case studies. Results show that *CMSU* is complete and correct. Second, we assessed the applicability of *MoSH* (the proposed UML profiles and modeling methodology) in terms of creating ETRMs for three case studies. We found that *MoSH* provides sufficient modeling constructs for modeling expected self-healing behaviors and environment uncertainties. However, applying *MoSH* needs 16% additional modeling effort on average.

As a proof of concept, we assessed the implementation of *TM-Executor* by testing a real-world case study with a random test strategy. We assessed the performance of *TM-Executor* in terms of time required to perform various testing steps such as generating test data and validating state invariants (test oracles). Results show that time taken by *TM-Executor* is very small, i.e., in the order of milliseconds. In addition, while testing the system with the random

test strategy, we managed to find one fault in its self-healing behavior in the presence of uncertainties. Note that such a fault can only be found in the presence of uncertainties.

1.5 Structure of the Paper

The rest of this paper is organized as follows: Section 2 presents a running example. Section 3 presents the conceptual model, i.e., *CMSU*, whereas *MoSH* profiles and its associated modeling methodology are explained in Section 4. Section 5 describes *TM-Executor*, followed by the evaluation in Section 6. Related work is discussed in Section 7 and we conclude the paper in Section 8.

2. Running Example

In this section, we introduce a running example, which is an unmanned aerial vehicle control system—Remotely operated Aerial Model Autopilot (RAMA) from [13]. It consists of two subsystems (Figure 2): Ground Control Station (GCS) and Drone (quadcopter). A human pilot uses GCS to send movement instructions to the drone via the Micro Air Vehicle Communication Protocol (MAV Link). Based on the received movement instructions from the MAV link, estimated position from Position Location Unit (PLU), and terrain data from the Terrain Database, Navigation Unit (NU) calculates target pitch, yaw, roll and throttle instructions, and sends them to the four servos to make the drone perform expected movements.

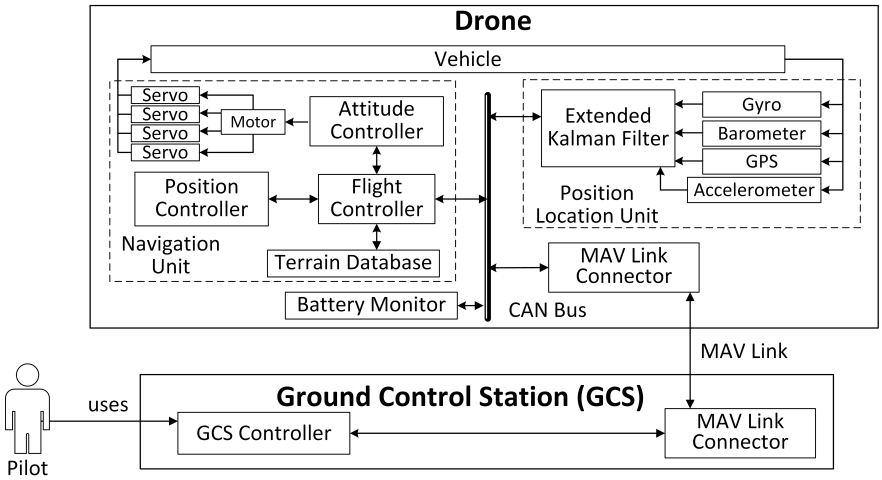

Figure 2 Key components and their Connections of RAMA

A key objective of the RAMA is to prevent the drone from crashing even if one or more components fail to work. To achieve this objective, the RAMA realizes a set of self-healing behaviors to handle faults during flight. For instance, if the MAV link between the drone and the GCS disconnects, the NU detects this error and the corresponding fault via the absence of heartbeats and automatically directs the drone to fly back to the launch location. Another example is the self-healing behavior for servo faults, that is, when one of the four servos stops

working, the fault is identified by comparing expected and actual attitudes of the drone and the NU can then choose to only control three dimensions (pitch, roll, and throttle), with the fourth dimension (yaw) uncontrolled, to maintain the flight.

Besides system component failures, environment uncertainties are another factor that has impacts on the operation of the RAMA. Such uncertainties include the accuracy of measurements and actuation, bandwidth and latency of the MAV link, and wind speed. To keep the flight stable, an adaptive control strategy has been implemented in the NU, which constantly adjusts control signals for the servos based on the drone's current attitude estimated by the PLU.

3. Conceptual Model for ShCPS and Uncertainty (*CMSU*)

To support EMBT of ShCPSs in the presence of uncertainties, we aim to provide a modeling framework (i.e., *MoSH*) including a set of UML profiles that allows creating ETRMs. The creation of UML profiles can be performed in two different ways as discussed in [14]: 1) directly creating a UML profile without developing a conceptual model first as how the UML profile for software architecture descriptions (ADL) was developed [15], and 2) creating a conceptual model and then defining a UML profile based on the conceptual model, which is more systematic and rigorous as recommended by Bran Selic in [14]. The second approach has been applied to develop the UML Profile for Modeling of Real-Time and Embedded Systems (MARTE) [16]. We opted for the second way; we first created the Conceptual Model for ShCPS and Uncertainty (*CMSU*), followed by creating corresponding UML profiles (Section 4).

In the rest of the section, we present *CMSU* in three parts: the self-healing CPS conceptual model (Section 3.1), the self-healing behavior conceptual model (Section 3.2), and the uncertainty conceptual model (Section 3.3).

3.1 The Self-Healing CPS Conceptual Model

The *Self-HealingCPS* conceptual model is presented in Figure 3 as a class diagram, whereas the concepts in the conceptual model are defined in Table 2. A *Self-HealingCPS* can be seen as a collection of heterogeneous, distributed and networked *PhysicalUnits* working together to control or monitor *PhysicalProcesses*, e.g., GCS and drone cooperating to control the flight process in the RAMA. Such a system can have its architecture being centralized, decentralized or hybrid. *Controller*s are the core elements of a *PhysicalUnit*, such as the control units in the running example. They provide the control logic and computation capabilities to the *PhysicalUnit* and communicate with other *Controller*s owned by other *PhysicalUnit*s via *Network*s. A *Controller* monitors and controls *PhysicalProcesse*s via *Sensor*s and *Actuator*s. Due to the stochastic nature of the *Environment*, events may occur in an uncertain manner,

which affects *PhysicalProcesses*. For example, in the RAMA, the wind direction and speed change constantly, which affects the flight process. Such changes are conceptualized as *Situations* in the conceptual model.

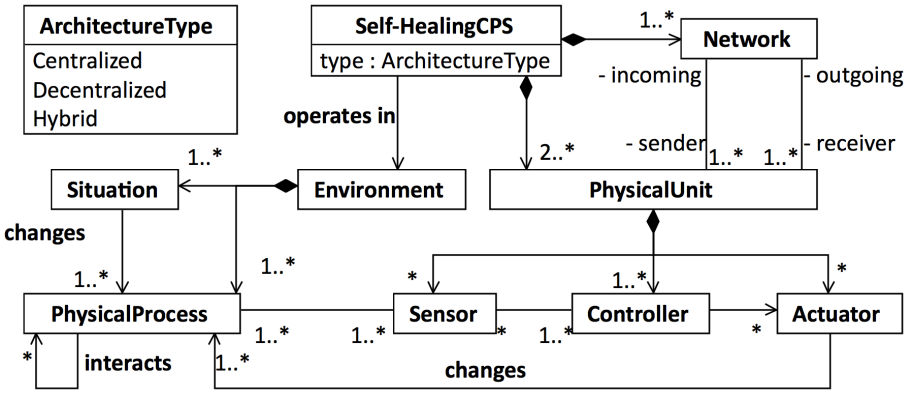

Figure 3 Self-Healing CPS Conceptual Model

Table 2 Concept Definitions of the Self-Healing CPS Conceptual Model

Concept	Definition
C1.Self-HealingCPS	A CPS, which can autonomously detect, diagnose and recover from *Faults* (C18)
C2.Environment	A physical world, where ShCPSs are situated.
C3.Situation	A set of *Environment* (C2) attributes and/or actions that affect *PhysicalProcesses* (C4) [17]
C4.PhysicalProcess	A sequence of chemical, physical, or biological activities for transport, storage of material, energy and etc. [18]
C5.PhysicalUnit	A physical device that can communicate with others, optionally having computation and control capabilities
C6.Network	The medium used as the communication channel among *PhysicalUnits* (C5)
C7.Sensor	A device that measures physical variables of a *PhysicalProcess* (C4)
C8.Actuator	A device that changes physical quantities of a *PhysicalProcess* (C4)
C9.Controller	A software deployed on a *PhysicalUnit* (C5), interacting with *Sensors* (C7) and *Actuators* (C8) either directly or indirectly, communicating with other *Controllers* and providing computational capability

3.2 The Self-Healing Behavior Conceptual Model

The *Self-HealingBehavior* conceptual model is presented in Figure 4 as a class diagram, and the concepts in the conceptual model are defined in Table 3. In a *Self-HealingCPS*, both hardware and software may have fault tolerance capabilities. For hardware, fault tolerance is typically achieved via introducing redundant hardware, which has limited adaptive capabilities at the runtime. In contrast, software can be reconfigured and modified, thus in the context of *Self-HealingCPS*, *Controllers* provide such self-healing capabilities, as shown in Figure 4.

Self-healing systems are defined by Debanjan Ghosh in [2] as "a self-healing system should recover from the abnormal ("unhealthy") state and return to the normative ("healthy") state, and function as it was prior to disruption". This requires a *Controller* to detect *Errors* in a timely fashion (via the *Self-Diagnosis* capabilities of its *Self-HealingBehaviors*) and react to the *Errors* to possibly restore its normal operation.

We used the standard definitions as defined by Avizienis et al. [19], "*Fault* is the cause of an *Error*". Meanwhile, "*Error* is a deviation of an external *State* from the correct one". Due to

*Fault*s, the system may fail to deliver correct service to its users. To avoid such situation, some *Controller*s with *Self-Diagnosis* capabilities are supposed to determine *Fault*s. Based on detected *Fault*s, recovery actions are executed to recover the system from the faults.

A *Controller* is equipped with *Probe*s and *Effector*s to make itself self-aware and adaptable. *Probe* and *Effector* are two types of interfaces that are used to inquire *Controller*'s *State*s and adjust *Controller*'s *Behavior*s respectively. In the RAMA, *Probe*s are monitoring interfaces that are used by the NU to constantly check variations of velocity and attitude. *Effector*s are the interfaces used to switch the control mode.

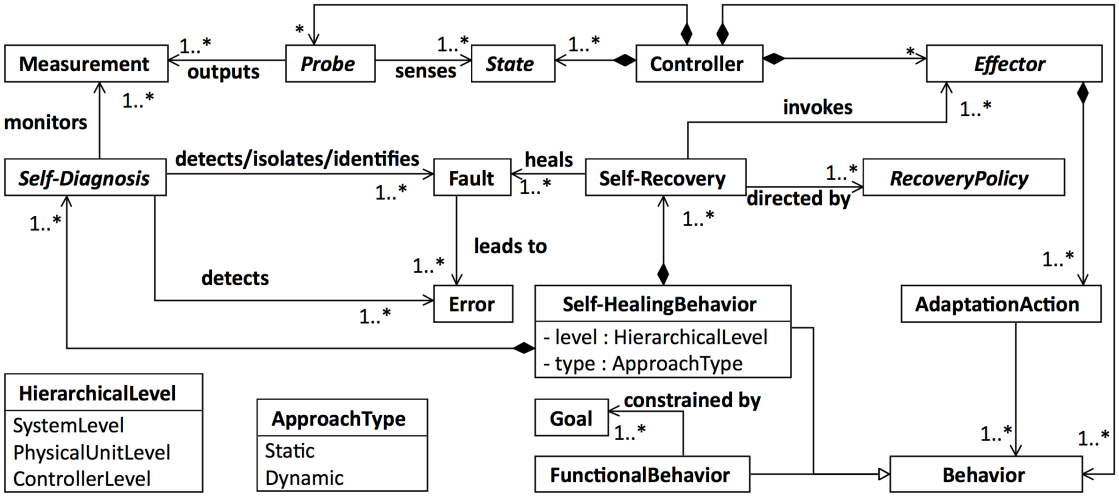

Figure 4 Self-Healing Behavior Conceptual Model (Overview)

Table 3 Concept Definitions of Self-Healing Behavior Conceptual Model

Concept	Definition
C10. Behavior	Describing a sequence of actions executed by a *Controller* (C9)
C11. FunctionalBehavior	The business logic of a *Controller* (C9) [20]
C12. Self-HealingBehavior	A sequence of *Self-Diagnosis* (C19) and *Self-Recovery* (C21) actions aiming at the recovery of a *Controller* (C9), a *PhysicalUnit* (C5) or the whole *Self-HealingCPS* (C1) from *Fault*s (C18)
C13. Goal	"A non-operational objective to be achieved by the composite system" [21]
C14. State	A particular combination of the attribute values of a *Controller* (C9) [22]
C15. Probe	A system measurement mechanism, which observes and measures the *State*s (C14) of a *Controller* (C9) [23]
C16. Measurement	A value of a *State* variable
C17. Error	"A deviation of an external *State* (C14) from the correct *State* (C14)"[24]
C18. Fault	"The cause of an *Error*" (C17) [19]
C19. Self-Diagnosis	The action that detects *Error*s (C17) and detects, isolates or identifies *Fault*s (C18), based on *Measurement*s (C16)
C20. RecoveryPolicy	A type of formal behavioral guide for *Self-Recovery* (C21) [25]
C21. Self-Recovery	The action that adapts the system for handling identified *Fault*s (C18) via *Effectors* (C22)
C22. Effector	A mechanism carrying out *AdaptationAction*s (C23) [23]
C23. AdaptationAction	A change in the system to accommodate *Fault*s (C18) [26]

As shown in Figure 4, a *Controller* has two types of *Behavior*s: 1) *FunctionalBehavior*s implementing business requirements of the system; 2) *Self-HealingBehavior*s that use *Probe*s and *Effector*s to monitor and maintain the correctness of *FunctionalBehavior*s. *Self-HealingBehavior*s are classified as static if they are fixed and pre-defined at the design time,

otherwise dynamic. Moreover, *Self-HealingBehavior*s are implemented at different hierarchical levels, e.g., healing only one function of the *Controller* (*ControllerLevel*), several functions of a *PhysicalUnit* (*PhysicalUnitLevel*), or the whole *Self-HealingCPS* (*SystemLevel*). For the RAMA case study, switching to the predefined control mode, in the case of hardware faults, is defined at the design time and thus the *Self-HealingBehavior* of the RAMA is static and implemented at *SystemLevel*.

One prerequisite of realizing *Self-HealingBehavior*s is the accurate specification of each component's *Goal*s in terms of its functional and/or extra-functional requirements. Moreover, *Goal*s at the system level can be decomposed into several sub-goals at the *PhysicalUnitLevel* and further at the *ControllerLevel*. Eliciting and specifying goals, which have been broadly studied in the requirements engineering community, are however out of the scope of this paper. For the RAMA, its essential goal is to guarantee the safe flight of the vehicle. Even if the core control unit is crashed, the RAMA should still keep the vehicle under control or at least safely land it.

A *Self-HealingBehavior* is composed of two functionalities: *Self-Diagnosis* in charge of detection, isolation, or identification of faults, and *Self-Recovery* responsible for recovering the system from faults. First, a *Self-Diagnosis* behavior evaluates states of *Controller*s according to *Measurement*s collected via *Probe*s. If the *Measurement*s deviate from expected values, violate constraints, or match a pattern of a fault, it means that the states of the *Controller*s have deviated from the correct ones. In this way, a *Self-Diagnosis* behavior can detect the occurrence of faults, or isolate the location of the faults, or even identify the magnitude of the faults. Afterward, a *Self-Recovery* behavior is alerted to react to the detected faults. Directed by *RecoveryPolicies*, the *Self-Recovery* behavior determines how to adapt the system to the faulty condition via *Effector*s. The following subsections further explain *Self-Diagnosis* and *Self-Recovery*.

3.2.1 The Self-Diagnosis Conceptual Model

Figure 5 presents the *Self-Diagnosis* conceptual model, and the concepts in the conceptual model are defined in Table 4. The *Self-Diagnosis* behavior, the fundamental part of a *Self-HealingBehavior*, constantly detects *Fault*s from a set of *Measurement*s. The detectability of a diagnosis behavior can be classified into three levels: *FaultDetection*, *FaultIsolation*, and *FaultIdentification* [27]. The diagnosis at the *FaultDetection* level can only detect the occurrence of faults; the *FaultIsolation* level diagnosis can determine which kind of faults has happened, and the diagnosis at the *FaultIdentification* level can deduce the magnitude of a fault. In the RAMA, the diagnosis behavior, determining if the MAV link between the drone and the GCS is disconnected, belongs to the *FaultDetection* level. The diagnosis of the servo

fault can identify the extent of the lift loss from a servo, thus belonging to the *FaultIdentification* level.

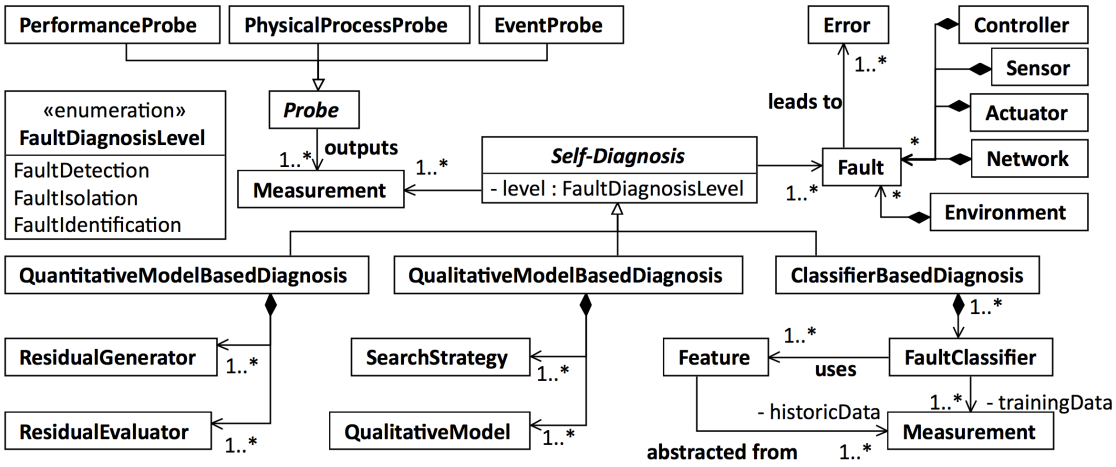

Figure 5 Self-Diagnosis Conceptual Model

A *Self-Diagnosis* behavior can be realized by three types of approaches. Two of them are achieved based on prior domain knowledge: *QuantitativeModelBasedDiagnosis* and *QualitativeModelBasedDiagnosis*; the other is derived from historic operational data of the system: *ClassifierBasedDiagnosis*.

For *QuantitativeModelBasedDiagnosis*, knowledge is expressed in a quantitative model, specifying mathematical relations between inputs and outputs of the system [28]. Fault occurrences are determined by checking inconsistencies (residues) between actual outputs of the system and expected outputs calculated from the model, via *ResidualGenerator* and *ResidualEvaluator*. In contrast, *QualitativeModelBasedDiagnosis* expresses domain knowledge in a *QualitativeModel*, i.e., qualitative relations between different system elements [28], such as cause-effect relations and fault trees. With this kind of approach, the actual execution of the system is checked against the *QualitativeModel*, directed by *SearchStrategies*, to realize fault diagnosis. The third type of diagnosis—*ClassifierBasedDiagnosis*, utilizes historical execution data (*Measurement*s) to abstract quantitative and qualitative *Feature*s and construct *FaultClassifier*s, which are for classifying system states based on *Measurement*s.

*Probe*s supply various *Measurement*s to *Self-Diagnosis*, which are classified into *PerformanceProbe*s, *EventProbe*s, and *PhysicalProcessProbe*s, based on types of *Measurement*s they provide. *PerformanceProbe*s are responsible for monitoring system's performance such as response time, throughput and availability. *EventProbe* monitors a *Controller*'s behavior described as a trace of events such as function calls and exceptions. The state of *PhysicalProcess*es can be accessed through *PhysicalProcessProbe*s such that a *Self-Diagnosis* behavior can decide if a *PhysicalProcess* is proceeding as expected. For the RAMA case study, all the three kinds of *Probe*s are used to detect faults, including monitoring the interface of the state update time (*PerformanceProbe*) for detecting the disconnected radio

control channel, interfaces for discovering unhealthy sensors (*EventProbe*), and interfaces for obtaining actual locations of the vehicle (*PhysicalProcessProbe*) for catching abnormal navigation behaviors.

Table 4 Concept Definitions of Self-Diagnosis Conceptual Model

Concept	Definition
C24. QuantitativeModelBasedDiagnosis	A *Self-Diagnosis* (C19) method based on mathematical models of the system [28]
C25. ResidualGenerator	A module for calculating inconsistencies between *Measurement*s (C16) and expected values computed from mathematical models of the system
C26. ResidualEvaluator	A module for determining the normal range of the residual
C27. QualitativeModelBasedDiagnosis	A *Self-Diagnosis* (C19) method based on qualitative causal models or abstraction hierarchies of the system [28]
C28. QualitativeModel	Qualitative causal models or abstraction hierarchies of the system, representing qualitative relations among different elements in the system
C29. SearchStrategy	A search method for defining how to locate a *Fault* (C18) in a system
C30. ClassifierBasedDiagnosis	A *Self-Diagnosis* (C19) method based on fault classifiers trained from historical data of the system [28]
C31. Feature	The attribute of a system behavior characterized by a combination of *Measurement*s (C16)
C32. FaultClassifier	A classifier for fault classification, which is built from historical data
C33. PerformanceProbe	A *Probe* (C15) for monitoring system's performance
C34. EventProbe	A *Probe* (C15) for monitoring events occurred in *Self-HealingCPS* (C1)
C35. PhysicalProcessProbe	A *Probe* (C15) for monitoring the state of a *PhysicalProcess* (C4)

3.2.2 The Self-Recovery Conceptual Model

Figure 6 presents the conceptual model of *Self-Recovery*, and the concepts in the conceptual model are defined in Table 5. After a fault has been detected by a *Self-Diagnosis* behavior, a *Self-Recovery* behavior decides which *AdaptationAction*(s) to take to handle the fault, directed by *RecoveryPolicies*. *Effector*s provide *Self-Recovery* behaviors the basis to modify and heal the system in case of faults. According to types of modified elements, *Effector*s can be classified into three types: *ParameterEffector*s for adjusting system components' parameters [29], *ArchitectureEffector*s for adding, removing, or replacing system components [30], and *ControlEffector*s for changing *FunctionalBehavior*(s) of a *Controller* in response to faulty conditions.

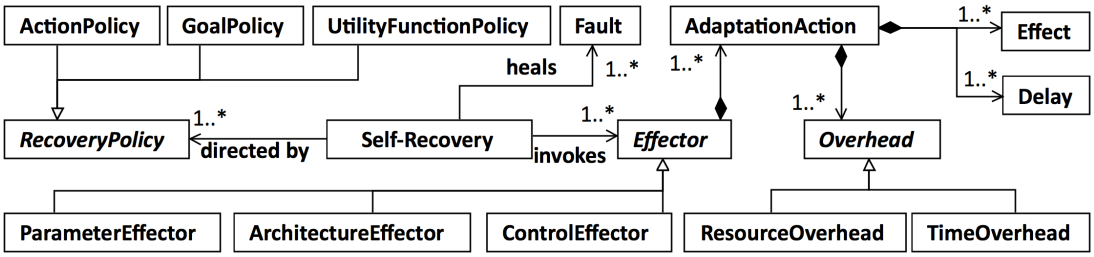

Figure 6 Self-Recovery Conceptual Model

Each *AdaptationAction* of *Effector*s can be considered as a variation point of the system, which enables reconfiguration and adaptation at runtime. Different *AdaptationAction*s have different *Effect*s on the system and may have different *Overhead*s and *Delay*s. A *Self-Diagnosis* behavior has a trade-off between adaptation benefits and costs in terms of time and/or resource consumption (i.e., *TimeOverhead* and *ResourceOverhead*).

In the literature, there are three types of *RecoveryPolicy* [25]: *ActionPolicy*, *GoalPolicy,* and *UtilityFunctionPolicy*. *ActionPolicy* can be seen as a pair in the form of <condition, action>. If the condition is satisfied, then a corresponding action is executed [31], which is applied in the RAMA case study. *GoalPolicy* specifies desired states, which requires a sequence of *AdaptationAction*s to be taken to make the system transit from a faulty state to the desired one [32]. *UtilityFunctionPolicy* defines an objective function containing multiple objectives of the system to guide the system to move towards the desired state in terms of utility values [33].

Table 5 Concept Definitions of Self-Recovery Conceptual Model

Concept	Definition
C36. Delay	Time interval between the initiation and completion of an *AdaptationAction* (C23)
C37. Effect	Change of one or more *Behavior*s (C10) caused by an *AdaptationAction* (C23)
C38. Overhead	Overhead for executing *AdaptationAction*s (C23)
C39. ResourceOverhead	Resources required for executing *AdaptationAction*s (C23)
C40. TimeOverhead	Time required for executing *AdaptationAction*s (C23)
C41. ActionPolicy	Specifying which *AdaptationAction*(s) (C23) should be taken for handling a *Fault* (C18) [25]
C42. GoalPolicy	Specifying desirable *State*s (C14) of the system and target states of *Self-HealingBehaviors* (C12) [25]
C43. UtilityFunctionPolicy	Assigning each *State* (C14) of the system a utility value, which directs the system moving towards a state with a higher utility value [25]
C44. ParameterEffector	An *Effector* (C22) for changing parameter values
C45. ArchitectureEffector	An *Effector* (C22) for updating system architecture
C46. ControlEffector	An *Effector* (C22) for modifying the control mode of a *Controller* (C9)

3.3 The Uncertainty Conceptual Model

Uncertainty is intrinsic in *Self-HealingCPS* due to the tight integration of *PhysicalProcess*es, *FunctionalBehavior*s, and *Self-HealingBehavior*s. Therefore, *Uncertainties* should be studied and analyzed in order to establish confidence that a *Self-HealingCPS* can eventually deal with *Uncertainties* in a graceful manner during its operation. In this section, we provide a general conceptual model to understand, classify, and characterize *Uncertainties* for the purpose of testing *Self-HealingCPS*s in the presence of *Uncertainties*. The *Uncertainty* conceptual model is presented in Figure 7 and the concepts in the conceptual model are defined in Table 6. In Table 7, we present a simple example to illustrate the conceptual model.

Our definition of *Uncertainty* conforms to the definition provided by Walker et al. [7]: "limited knowledge about future, past, or current events". We adapt this definition to the context of testing as the lack of knowledge about the value of an *UncertainFeature* (C47) at a given point of time during a testing process. For instance, for the RAMA, the actual value of the packet loss rate (*UncertainFeature*) constantly varies from 0% to 100% during testing. Thus at a given point of time, the value of packet loss rate is uncertain. Here the given point of time during testing is conceptualized as a *TimeInstance* (Figure 7).

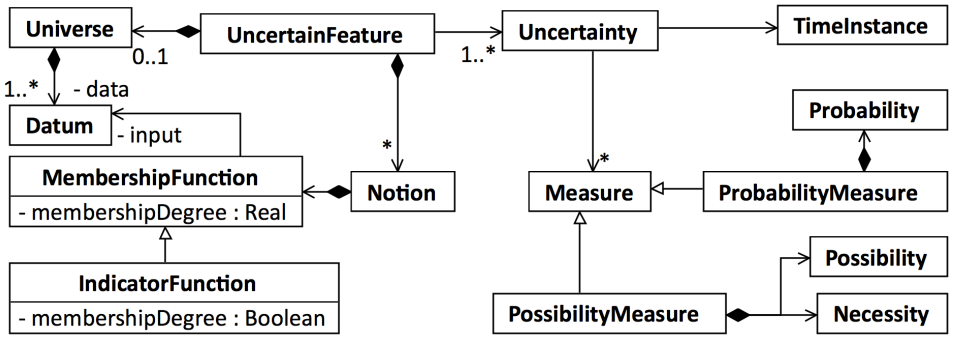

Figure 7 Uncertainty Conceptual Model

Table 6 Concept Definitions of Uncertainty Conceptual Model

Concept	Definition
C47. UncertainFeature	A feature whose value is uncertain due to lack of knowledge
C48. Uncertainty	Lack of knowledge about the value of an *UncertainFeature* (C47) at a given *TimeInstance* (C49)
C49. TimeInstance	A point of time during a testing process
C50. Universe	The set of all potential values of an *UncertainFeature* (C47)
C51. Notion	A qualitative description of an *UncertainFeature* (C47)
C52. Datum	One possible value of an *UncertainFeature* (C47)
C53. MembershipFunction	A function defining the degree of belonging to a *Notion* (C51)
C54. IndicatorFunction	A binary belonging function with 1 for elements belonging to the *Notion* (C51), 0 for other elements
C55. Measure	Measuring the likelihood of a *Datum* (C52) or *Notion* (C51), related to an *Uncertainty* (C48)
C56. ProbabilityMeasure	A *Measure* (C55) of uncertainty using *Probability* (C57) to characterize the likelihood
C57. Probability	Quantifying the chance that an event will occur [34]
C58. PossibilityMeasure	A *Measure* (C55) of uncertainty using *Possibility* (C59) and *Necessity* (C60).
C59. Possibility	Describing the plausibility that an event will occur [35]
C60. Necessity	Describing the credibility that an event will occur [35]

Universe describes a set that contains all values that an *UncertainFeature* may take. Typically, an *UncertainFeature* should have one *Universe*; however, in certain cases, it is possible that we do not have sufficient knowledge about the *Universe* and cannot specify it. A value of the *UncertainFeature* is defined as a *Datum* (C52). Taking the packet loss rate for example, the *Universe* of the *UncertainFeature* is an interval from 0% to 100%, and every value within this interval is a *Datum*.

In certain cases, values of an *UncertainFeature* can only be described in a qualitative manner. As presented in Table 7, the packet loss rate of the MAV link is described as low, medium and high, which are represented as *Notion*s (C51) in the conceptual model. A *Notion* has one *MembershipFunction* (C53), which determines the extent to which a *Datum* belongs to the *Notion*. More specifically, a *MembershipFunction* of a *Notion* takes one *Datum* as input and outputs a real value between 0 and 1, representing the membership degree of the *Datum* to the *Notion*. This means a *Datum* could partially belong to multiple *Notion*s, and in this case,

each *Notion* is a fuzzy set[1]. *IndicatorFunction* (C54) is a specialized *MembershipFunction*, which only outputs 0 or 1, meaning that a *Datum* either belongs or does not belong to the *Notion* associated with the *IndicatorFunction*. In this case, the corresponding *Notion* is a crisp set[2]. Table 7 shows an example for both of the cases. When using the *IndicatorFunction*, the boundaries of the three *Notion*s (i.e., Low, Medium and High) are crisp, i.e., a packet loss rate only belongs to one of the three *Notion*s. In contrast, the boundaries defined by the *MembershipFunction* are fuzzy. In this case, a packet loss rate of 0.02 belongs to Low with 50% membership degree, to Medium with 45% membership degree, and to High with 5% membership degree.

Table 7 An Example of Uncertainty

Concepts	**Example**
UncertainFeature	Packet loss rate of the MAV link
Uncertainty	Actual value of the packet loss rate at a given time instance
Universe	The interval from 0% to 100%
Datum	$\forall x, x \in [0\%, 100\%]$
Notion	$Low, Medium, High$
MembershipFunction	$M_{Low}(x) = 1/(1 + e^{100(x-0.02)})$ $M_{Medium}(x) = 1/(1 + e^{100(x-0.05)}) - 1/(1 + e^{100(x-0.02)})$ $M_{High}(x) = 1/(1 + e^{100(0.05-x)})$
IndicatorFunction	$M_{Low}(x) = \begin{cases} 1, x \in [0, 0.02) \\ 0, x \notin [0, 0.02) \end{cases}$ $M_{Medium}(x) = \begin{cases} 1, x \in [0.02, 0.05) \\ 0, x \notin [0.02, 0.05) \end{cases}$ $M_{High}(x) = \begin{cases} 1, x \in [0.05, 1] \\ 0, x \notin [0.05, 1] \end{cases}$

An *Uncertainty* may be measured with different *Measure*s (C55). From complete certainty to total ignorance, there exist five intermediate levels as defined in [7]. Table 8 shows these five levels along with their relations to *Measure*, *Datum*, and *Notion*.

For Level 1 *Uncertainty*, at a given *TimeInstance*, the value of an *UncertainFeature* is one value with a margin of error. In other words, one is absolutely certain that the value falls within this margin. For this reason, no qualitative specification (*Notion*) is required for this level. In the running example, a servo's maximum thrust could be determined according to its product specification. However, this value is not accurate, and a tolerance interval is given to specify the range of the value. Therefore, the maximum thrust belongs to Level 1 *Uncertainty*, i.e., an absolute value with a margin of error.

Level 2 *Uncertainty* stands for the situation that an *UncertainFeature* has alternate values with *Probabilities* (C57). Thus, *ProbabilityMeasure* (C56) is used at this level to map every *Datum* or *Notion* to a *Probability*. For instance, the measurement error of a GPS in the RAMA

[1] "The fuzzy set is defined mathematically by assigning a degree of membership to each possible value in the universe of discourse." [35]
[2] "The crisp set is defined as a set that dichotomize the individuals in a universe of discourse into two groups: members and nonmembers." [35]

is an *Uncertainty* conforming to a normal distribution. Through statistical analyses, the normal distribution can be determined, and thus it is a Level 2 *Uncertainty*.

Similarly, for Level 3 *Uncertainty*, the *Probability* of each possible value is unknown, but each possible value is bound with a ranked likelihood, which can be specified via *Possibility* (C59) or *Necessity* (C60) of the *PossibilityMeasure* (C58). Following the running example, due to the limited knowledge, probability distributions of wind speed and direction cannot be determined, and we can only compare the likelihoods of different potential values. Consequently, *PossibilityMeasure*s are used to specify the *Measure* of Level 3 *Uncertainty*. For instance, the likelihoods of low, medium, and high wind speed are little, large, and little respectively. Accordingly, their possibilities can be specified as 0.2, 0.7, and 0.2 to reflect their ranked likelihood.

A Level 4 *Uncertainty* is the case when one is able to enumerate multiple alternative values of an *UncertainFeature* but cannot rank their likelihoods, due to for example a lack of knowledge, or disagreements among modelers [7]. At last, Level 5 *Uncertainty* represents situations that what is known is only that we do not know (i.e., *known unknowns*). In other words, the ignorance (the "unknowns" part of *known unknowns*) is recognized (the "known" part of *known unknowns*). More specially, neither *Universe* nor *Measure* of an *Uncertainty* of an *UncertainFeature* at this level is known. The only thing known is the existence of the *UncertainFeature*. For Level 4 and Level 5 *Uncertainties*, knowledge about them is too little to explicitly model them to be useful for enabling EMBT, and they are excluded from our modeling methodology.

Table 8 Uncertainty Levels

Level	Datum	Measure	Notion
Complete Certainty	A datum	N/A	N/A
Level 1	A determined datum with a margin of error	N/A	N/A
Level 2	A set of data	Probability Measure	Related
Level 3	A set of data	Possibility Measure	Related
Level 4	A set of data	N/A	Related
Level 5	Known Unknowns	N/A	N/A
Total Ignorance	Unknown Unknowns	N/A	N/A

4. The *MoSH* Modeling Notations and Methodology

Based on *CMSU* presented in Section 3, we develop *MoSH*, which comprises of four UML profiles and a modeling methodology, for enabling the development of ETRMs to facilitate EMBT of ShCPSs. An overview of the *MoSH* modeling framework is presented in Figure 8, where it shows that the *MoSH* modeling notations consist of four UML profiles: ShCPS Component Profile, ShCPS Behavior Profile, ShCPS Uncertainty Profile, and ShCPS Testing Profile.

In addition, the *MoSH* methodology (an overview of which is presented in Figure 9) provides a step-wise procedure for creating ETRMs. Each of the four high-level steps

corresponds to the application of each *MoSH* profile and these steps are introduced together with the profiles in the following subsections.

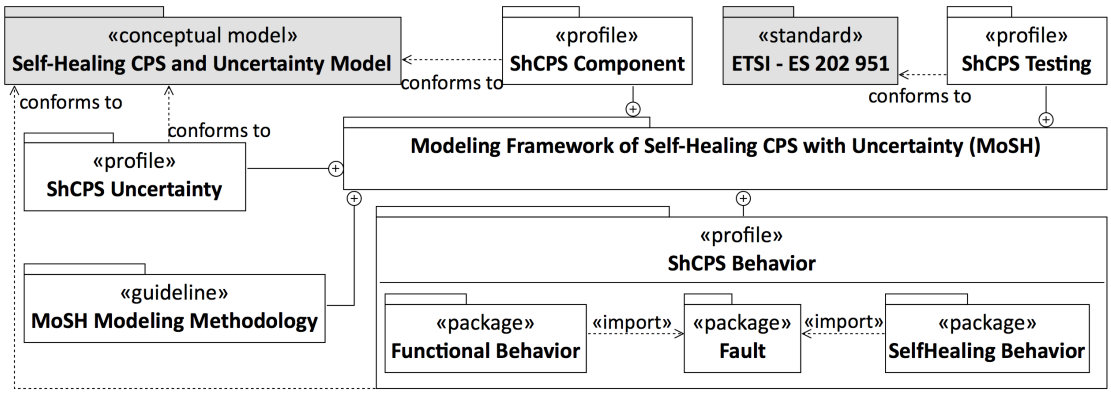

Figure 8 Overview of the *MoSH* Modeling Framework

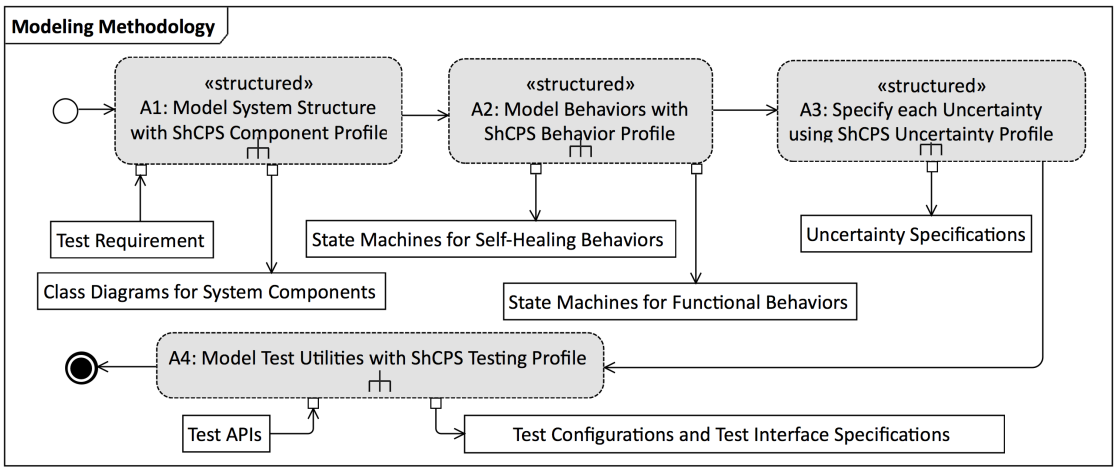

Figure 9 Overview of the *MoSH* Modeling Methodology

4.1 Model System Structure with ShCPS Component Profile

4.1.1 ShCPS Component Profile

The ShCPS Component profile captures key components of a ShCPS (C1). A ShCPS is comprised of a set of physical units (C5) cooperating together via heterogeneous networks (C6). Sensors (C7), actuators (C8) and controllers (C9) constitute the major components of each physical unit. Accordingly, six stereotypes are defined for these concepts, shown in Table 9. All the six stereotypes extend UML metaclass *BehavioredClassifier* as they all realize intended behaviors.

«Network» represents communication channels among physical units. The "uncertainty" attribute of «Network» captures indeterminate feature of a network such as bandwidth, latency, and packet loss rate. These features can be modeled as attributes stereotyped with «Uncertainty» to specify the state of knowledge of an uncertain feature (Section 4.3.2).

Depending on the visibility of interfaces, a physical unit can be seen as a black box, if only external interfaces are accessible, or a white box, if the implementation of its control software

is accessible. For the first case, a physical unit can be modeled as a *BehavioredClassifier* stereotyped with «PhysicalUnit». Besides the "uncertainty" representing uncertain features of a physical unit, a «PhysicalUnit» has zero to many goals, which are specified as constraints that should be obeyed by the classifier. For the second case, a physical unit can be decomposed into sensor, actuator, and controller classifiers, stereotyped with «Sensor», «Actuator», and «Controller» respectively. The "uncertainty" attribute of «Sensor» and «Actuator» captures uncertain features related to the accuracy of a measurement or an actuation, including the additive error (bias), multiplicative error (scale), stochastic error (noise), and temporal error (latency).

«PhysicalProcess» (C4) is an abstract concept and its "uncertainty" attribute specifies uncertainties arising from mathematical equations of physical variables in the physical process, and uncertain parameter values in the equations.

Table 9 Stereotypes in ShCPS Component Profile

Stereotype	Metaclass	Attribute
«SelfHealingCPS»	Package	type: ArchitectureType
«Network»	BehavioredClassifier	uncertainty: Uncertainty [*]
«PhysicalUnit»	BehavioredClassifier	goal : Constraint [*] uncertainty: Uncertainty [*]
«Sensor»	BehavioredClassifier	uncertainty: Uncertainty [*]
«Actuator»	BehavioredClassifier	uncertainty: Uncertainty [*]
«Controller»	BehavioredClassifier	goal : Constraint [*] uncertainty: Uncertainty [*]
«PhysicalProcess»	BehavioredClassifier	uncertainty: Uncertainty [*]

4.1.2 Model System Structure (A1)

The first step of building an ETRM is to capture the structure of the ShCPS under test (SUT) using ShCPS Component Profile. The modeling process is summarized in Figure 10. First, the physical processes, physical units, and networks, which constitute the SUT, are captured as separate classes, stereotyped with «PhysicalProcess», «PhysicalUnit», and «Network» respectively. Physical units can be further decomposed into sensors, actuators, and controllers, each of which is specified as a class stereotyped with «Sensor», «Actuator», or «Controller». Figure 11 shows a partial structural model of the RAMA, which consists of two physical units (*GroundControlStation*, *Drone*) connected through a network (*MAVLink*). Since the *GroundControlStation* is mainly used for user's input/output and is not the focus of testing, its internal components are not captured in this model. On the other hand, the *Drone* is decomposed into several controllers, sensors, and one actuator to more explicitly specify the expected behaviors of the *Drone*.

All accessible state variables that can be queried by testing interfaces are specified as class attributes, such as *mode* of *NavigationUnit* and *throttle* of *Motor*, as shown in Figure 11. Operations and signal receptions denote testing interfaces provided by corresponding components, including output operations for querying state variables, input operations for

manipulation, and fault injections for introducing faults to trigger self-healing behaviors, which are stereotyped with «OutputOperation», «InputOperation», and «FaultInjection» (defined in ShCPS Testing profile, Section 4.4.1). As presented in Figure 11, every class has one or more operations for monitoring or controlling the corresponding component. Two fault injection operations (*disconnect()* of *MAVLink* and *disableGPS()* of *GPS*) are also implemented to simulate two faults: disconnection from the *GroundControlStation* and loss of *GPS* signals respectively.

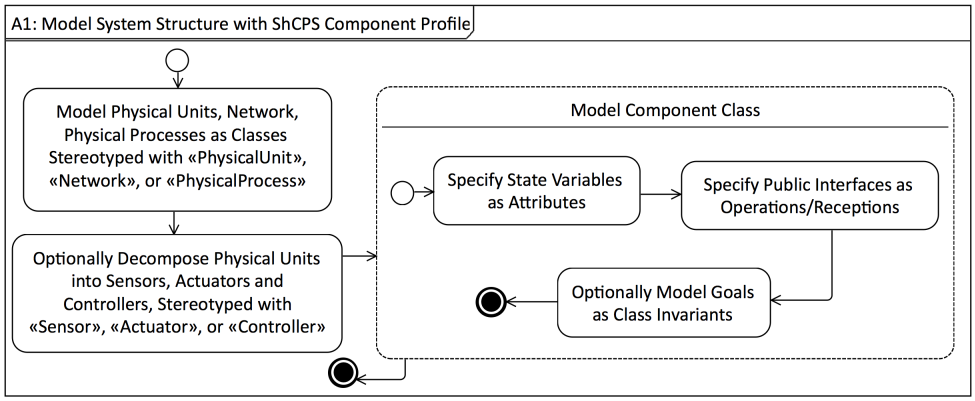

Figure 10 Model System Structure

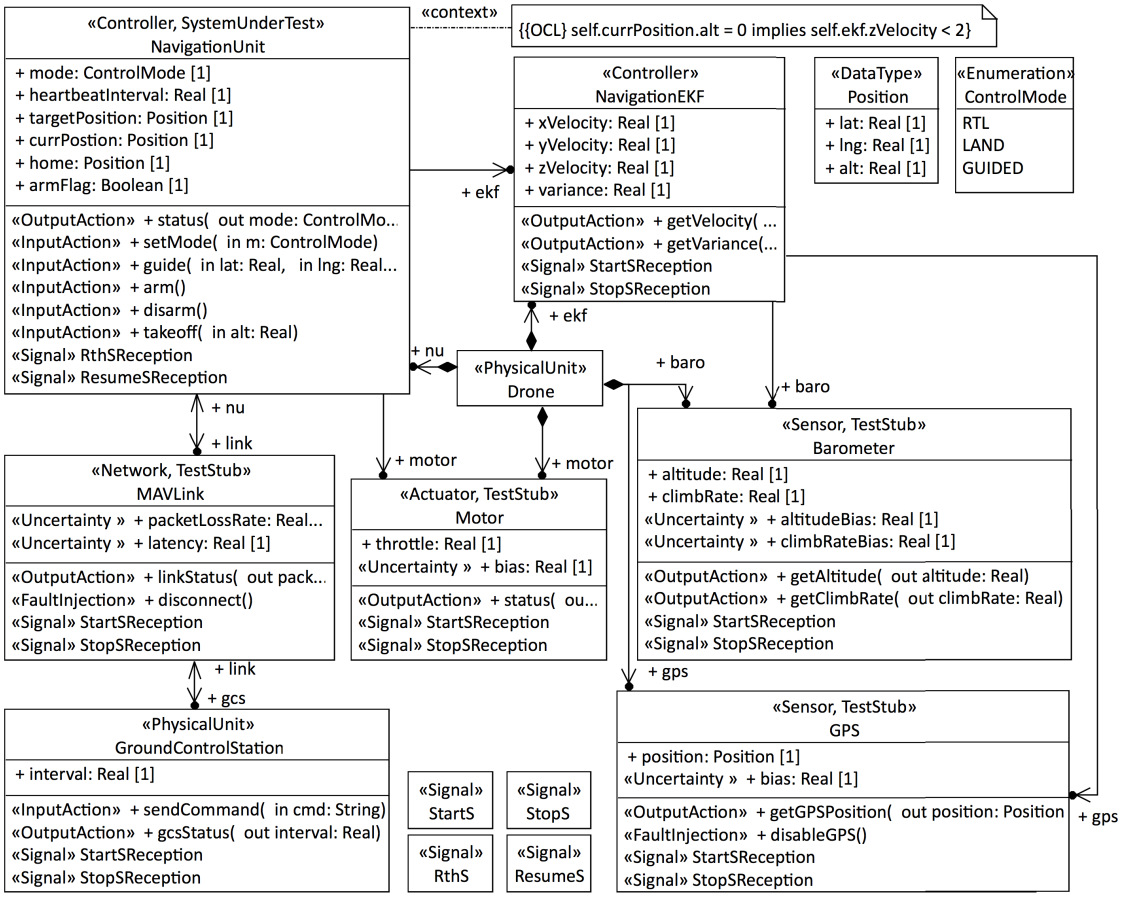

Figure 11 Structural Model of RAMA (Partial)

By assigning the value of each stereotype attribute, testers can systematically specify goals of these components. Goals of physical units and controllers, defined as OCL (Object

Constraint Language) constraints, represent functional and/or extra-functional requirements that should always be satisfied by the physical units and controllers. All the constraints are defined on class attributes such that they can be validated based on attributes' values obtained via testing interfaces. One goal of *NavigationUnit* is shown in Figure 11, which is about avoiding a crash on the ground, i.e., when *Drone* lands on the ground (*self.currPosition.alt = 0*), its vertical velocity should be below 2 meters per second (*self.ekf.zVelocity < 2*).

4.2 Model Behaviors with ShCPS Behavior Profile

4.2.1 ShCPS Behavior Profile

ShCPS Behavior Profile is proposed to specify expected self-healing behaviors (C12) of a ShCPS for the purpose of enabling EMBT. Since the objective of self-healing behaviors is to recover functional behaviors (C11) from faults (C18), the expected functional behaviors should also be captured to assess the utilities of self-healing behaviors. This profile has three packages: *Fault*, *Functional Behavior*, and *SelfHealing Behavior*, as shown in Table 10. The *Functional Behavior* and *Fault* packages provide the capability of modeling functional behaviors with potential faults whereas the *SelfHealing Behavior* package is for specifying the process of fault diagnosis (C19) and recovery (C21).

To capture normal and faulty states of a ShCPS, state machines are chosen to specify expected functional behaviors stereotyped with «FunctionalBehavior». Potential faults influencing these behaviors are hardware and software crashes, which are characterized by abnormal output values or unresponsiveness. «Fault» extending UML metaclass *ChangeEvent* is used to represent the occurrence of a potential fault and the change expression of the *ChangeEvent* defines the condition, under which the fault is regarded as having occurred. As shown in Figure 12, an occurrence of the disconnection of *MAVLink* is a potential fault in the RAMA, which is specified as a *ChangeEvent*, whose change expression is "latency > 3". This means that a disconnection fault occurs if the delay of *MAVLink* exceeds 3 seconds, which requires *NavigationUnit* to perform self-healing behaviors to keep the flight normal.

The "injectionOperation" attribute of «Fault» specifies a specialized testing interface for simulating the occurrence of a fault. Consequent states are stereotyped with «Error» (C17) to be distinguished from normal states. Following the example in Figure 12, *disconnect()* is a fault injection interface for simulating disconnection fault, which makes *MAVLink* enter an error state *Disconnected*, stereotyped with «Error».

Due to limited knowledge, a functional behavior may be indeterminate, which should be captured in a non-determinate state machine. «UncertainState» represents an indeterminate fragment of a functional behavior, a state with multiple outgoing transitions that have the same triggers and guards but different target states. Its "uncertainOutgoing" attribute specifies uncertainties caused by this indeterminism.

Table 10 Stereotypes in ShCPS Behavior Profile

Package	Stereotype	Metaclass	Attribute
Fault	«Fault»	ChangeEvent	name: String injectionOperation: FaultInjection [*]
	«Error»	State	name: String
Functional Behavior	«FunctionalBehavior»	StateMachine, Region	fault: Fault [*] error: Error [*]
	«UncertainState»	State	uncertainOutgoing: Uncertainty [1..*]
SelfHealing Behavior	«SelfHealingBehavior»	StateMachine, Region	type: ApproachType level: HierarchicalLevel
	«Monitoring»	State, StateMachine	measurement : Property [1..*]
	«FaultIdentification»	State, StateMachine	fault: Fault [1..*]
	«Adaptation»	State, StateMachine	name: String

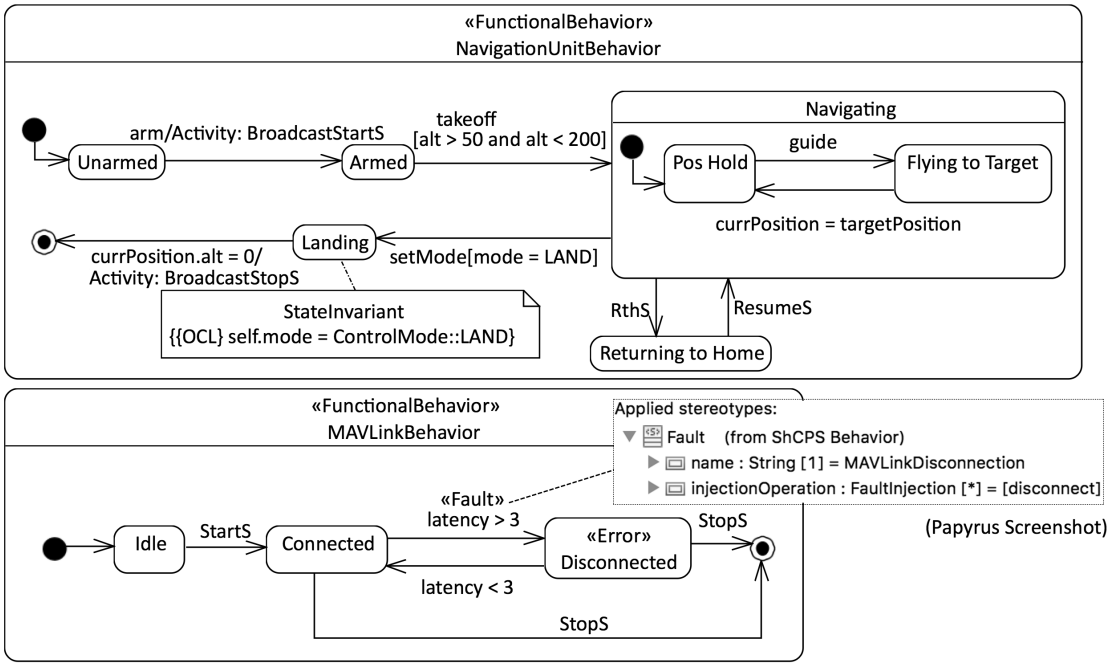

Figure 12 Examples of Functional Behaviors

Self-healing behaviors are captured in separate state machines stereotyped with «SelfHealingBehavior» and three stereotypes (i.e., «Monitoring», «FaultIdentification», and «Adaptation») are defined for annotating states in these behaviors. Figure 13 shows an example of self-healing behaviors, which specifies how *NavigationUnit* handles the disconnection fault.

After passing the initial state, a self-healing behavior enters a «Monitoring» state, where the system constantly checks various measurements (C16) from performance probes (C33), event probes (C34), and physical process probes (C35), as illustrated in Section 3.2.1. Monitored measurements are captured in the "measurement" attribute of «Monitoring». Notice that, there are two kinds of monitoring in a ShCPS. One is about monitoring environments through sensors and the other is about detecting errors and faults via probes. Since our focus is on self-healing behaviors, we only explicitly capture the second kind of monitoring in ETRMs. Therefore, we included monitoring as part of «SelfHealingBehavior».

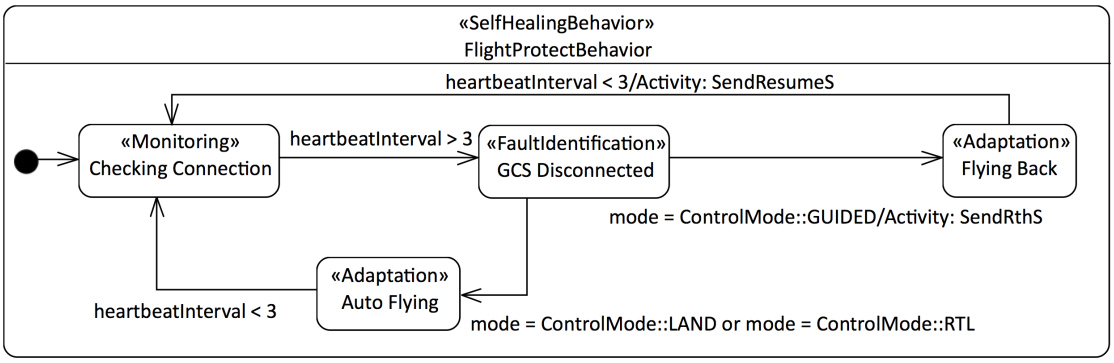

Figure 13 An Example of Self-Healing Behavior

Fault diagnosis logics are captured as transitions, originating from a «Monitoring» state and terminating on a «FaultIdentification» state. Triggers of transitions are specified as *ChangeEvent*s, whose change expressions define criteria for detecting faults. The self-healing behavior shown in Figure 13 uses *heartbeatInterval* to detect the disconnection of *MAVLink*. When the interval is over 3 seconds (*heartbeatInterval* > *3*), the self-healing behavior deems the fault occurred.

«Adaptation» is used for annotating adaptations (C23) that are used by a self-healing behavior to "heal" faults. A transition from a «FaultIdentification» state to an «Adaptation» state describe recovery policies (C20) specifying which adaptations are used to handle which faults. As shown in Figure 13, there are two ways to handle the disconnection fault. When *Drone*'s mode is *ControlMode::LAND* or *ControlMode::RTL*, no manual control is required to control the flight. In this case, *NavigationUnit* makes the *Drone* keep on its current task (the "*Auto Flying*" state). Otherwise, *NavigationUnit* changes the mode from *ControlMode::GUIDED* to *ControlMode::RTL* (via effect *SendRthS* of the transition from *GCS Disconnected* to *Flying Back*) under which the *Drone* flies back to where it takes off.

A transition from an «Adaptation» state to a «Monitoring» state indicates what to be done after a fault has been "healed". As shown in Figure 13, when the system is in the "*Flying Back*" state, as soon as the connection of *MAVLink* is rebuilt (i.e., *heartbeatInterval* < *3*), *NavigationUnit* changes the mode from *ControlMode::RTL* back to *ControlMode::GUIDED* to resume the original flight via effect *SendResumeS*.

4.2.2 Model Functional and Self-Healing Behaviors (A2)

The second step of developing an ETRM is to specify expected behavior for each class that has been identified in the first step (i.e., A1, Section 4.1.2). Figure 14 shows the two parallel processes of this step: modeling functional behaviors and modeling self-healing behaviors.

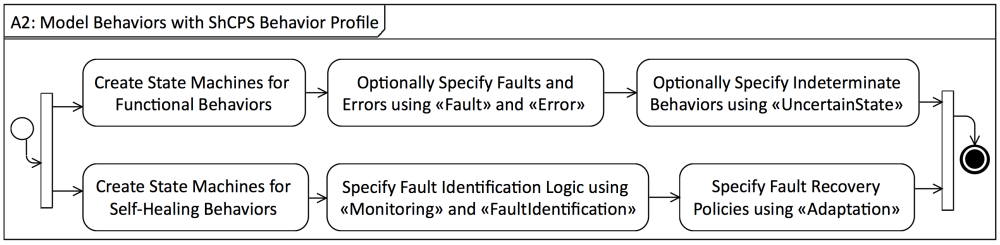

Figure 14 Model Functional and Self-Healing Behaviors

The functional behavior of a class is modeled as one or more state machines. Each state in the state machine should be precisely defined with state invariants, i.e., constraints on class attributes constructed in A1 (Section 4.1.2). For example, Figure 12 shows the state invariant of the *Landing* state, which is defined based on attribute *mode* of class *NavigationUnit* (Figure 11). With this invariant, one can test whether *NavigationUnit* is currently in the *Landing* state, according to the current value of *mode*. Any inconsistency between the actual state and the active state indicates a fault in the SUT.

In general, a transition between two states models a valid fragment of behavior [36], which can be triggered by a *CallEvent*, *SignalEvent*, or *ChangeEvent*. *CallEvent*s represent invocations from external systems or users via operational calls such as the transition between the *Armed* and *Navigating* states in Figure 12. Along with a *CallEvent*, a *Guard* (OCL constraint) can be specified to define the test data for invoking the operation corresponding to the *CallEvent*. *SignalEvent*s capture interactions among different state machines. Via sending signals in effects or state activities, firing a transition in one state machine can lead to transitions in other state machines being triggered. For example, the transition from the *Idle* state to the *Connected* state in the *MAVLinkBehavior* state machine will be triggered when the transition "*arm() / Activity: BroadcastStartS*" from the *Unarmed* state to the *Armed* state in the *NavigationUnitBehavior* state machine is activated (Figure 12). *ChangeEvent*s are used to model variations from internal components such as the transition from the "*Flying to Target*" state to the "*Pos Hold*" state in Figure 12.

A fault occurs in a system component; therefore, it is modeled as a *ChangeEvent*, a transition's trigger, which makes controllers enter an error state. The change expression of the event is defined based on test requirements. It defines under which condition the fault is regarded as occurred (Section 4.2.1). An indeterminate behavior can be specified as a special kind of state machine, where a state can have more than one outgoing transitions with the same trigger and guard. «UncertainState» is applied to annotate such states and the "uncertainOutgoing" attribute of the «UncertainState» is used to specify this uncertainty.

A self-healing behavior is also modeled as one or more state machines focusing on fault diagnosis and recovery. First, the logic of fault identification is specified via the transition between a «Monitoring» state and a «FaultIdentification» state. The «Monitoring» state represents the situation that no fault has been identified; while the «FaultIdentification» state

denotes that the self-healing behavior has identified a specific kind of a fault. The change expression of the *ChangeEvent*, triggering the transition between these two states, describes the criteria to detect the fault. As shown in Figure 13, the transition, from "*Checking Connection*" state to "*GCS Disconnected*" state, captures the logic of fault identification for the disconnection fault. The recovery policy performed by the self-healing behavior is modeled as the transition from a «FaultIdentification» state to an «Adaptation» state. The trigger of the transition is specified as a *ChangeEvent* whose change expression describes the adaptation condition (Figure 13). The trigger of the transition from an «Adaptation» state to a «Monitoring» state specifies the behavior after the fault has been successfully healed.

For both functional and self-healing behaviors, transition effects, and state entry, exit and doActivity behaviors can be used to specify interactions among state machines. According to the semantics of a Foundational Subset for Executable UML Models (fUML) standard [10], either an activity diagram or an opaque behavior with its method defined in the Action Language for fUML (ALF) [37], can be used to define an interaction behavior. For instance, effect *BroadcastStartS* of transition "*arm() / Activity: BroadcastStartS*" is defined as an activity diagram, as shown in Figure 15. The semantics of *ReadSelf* (defined in fUML and implemented in Moka) is to obtain an instance of *NavigationUnit* owning the *NavigationUnitBehavior* state machine. *BroadcastStartS* is a broadcast signal action, which is currently not defined in fUML and not implemented in Moka. We defined its execution semantics as follow: sending a signal to all instances of classes that are associated with the current class of the instance returned by *ReadSelf*. In our framework, we implemented the semantics of this action in Java and integrated it with the Moka framework. With this action, when the transition "*arm() / Activity: BroadcastStartS*" in the *NavigationUnitBehavior* state machine is fired, its effect broadcasts the *StartS* signal to *MAVLink*, *Motor*, and *NavigationEKF* (Figure 11). As a result, the transition *StartS* in the *MAVLinkBehavior* state machine will be fired, triggering *MAVLink* to enter the *Connected* state.

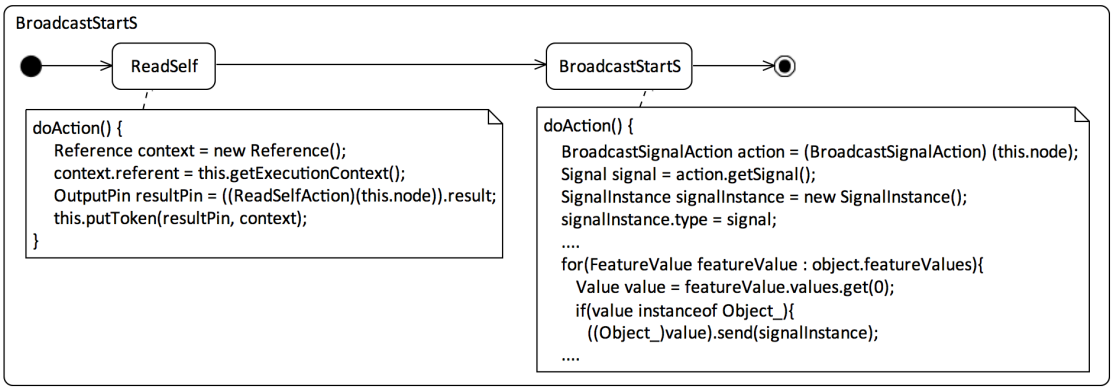

Figure 15 An Example of Transition's Effect Specified as an Activity Diagram

4.3 Specify Uncertainties using ShCPS Uncertainty Profile

4.3.1 ShCPS Uncertainty Profile

From the perspective of testing, an uncertainty (C48) is the *lack of knowledge about the value of an uncertain feature at a given point of time during the testing process*. As explained in Section 3.3, the state of knowledge of an uncertainty is specified by defining the universe (C50), notions of the uncertain feature (C51), and by stating the measure (C55) of the uncertainty. Accordingly, «Uncertainty» is defined, along with "universe", "notion", and "measure" attributes, as shown in Figure 16 (a). Each attribute corresponds to a newly defined datatype, which is introduced below.

The *Universe* datatype represents a collection of values. According to the type of value in universes, we derive 7 subtypes of the *Universe* datatype: *U_Boolean*, *U_Integer*, *U_Real*, *U_UnlimitedNatural*, *U_Transition*, *U_String*, and *U_Equation*. In addition, the numerical data types (*U_Integer*, *U_Real*, *U_UnlimitedNatural*) are further divided into intervals (*U_IntegerInterval*, *U_RealInterval*, *U_UnlimitedNaturalInterval*) and vectors (*U_IntegerVector*, *U_RealVetor*, *U_UnlimitedNaturalVector*). In total, 13 types of *Universe* are defined (Figure 16 (b)). The universe, specified as an interval, is a range with a minimum and a maximum bound, and a vector typed universe corresponds to a collection of values that are listed by its "item" attribute. As shown in Figure 17, attribute *altitudeBias* of class *Barometer* varies within a range and thus its universe is typed with *U_RealInterval*.

The *Notion* datatype is defined to specify a qualitative description of an «Uncertainty». As explained in Section 3.3, elements of a *Notion* are determined by a *MembershipFunction* (C53), which can be either a binary belonging function (*IndicatorFunction* (C54)) or a graded belonging function (*Gaussian*, *Sigmoidal*, *Triangle*, *Trapezoid*, *BellCurve*, *DiscreteFunction*). As shown in Figure 16 (d), typical *MembershipFunction*s are defined from the MATLAB fuzzy library [38] with their parameters captured in datatype attributes.

Depending on the state of knowledge of an «Uncertainty», three levels of uncertainty can be defined to quantify measures (Section 3.3). For Level 1 uncertainty, at a given point of time, the value of the uncertain feature is determined with a margin of error. The determined value and margin are captured in «Uncertainty»'s "universe" attribute. As shown in Figure 17 (Level 1), attribute *altitudeBias* of class *Barometer* is an uncertain feature, and its value is bounded by an error margin from -10 to 10.

For the other two levels of uncertainty, a determined value of the uncertain feature is unknown. Multiple values are possible to be true at a given point of time. If the probability of each value is known, the uncertainty belongs to the Level 2 uncertainty. The *ProbabilityMeasure* datatype is provided to specify the probability (C57) distribution of all

possible values, such as the *NormalDistribution* specified for the Level 2 uncertainty in Figure 17.

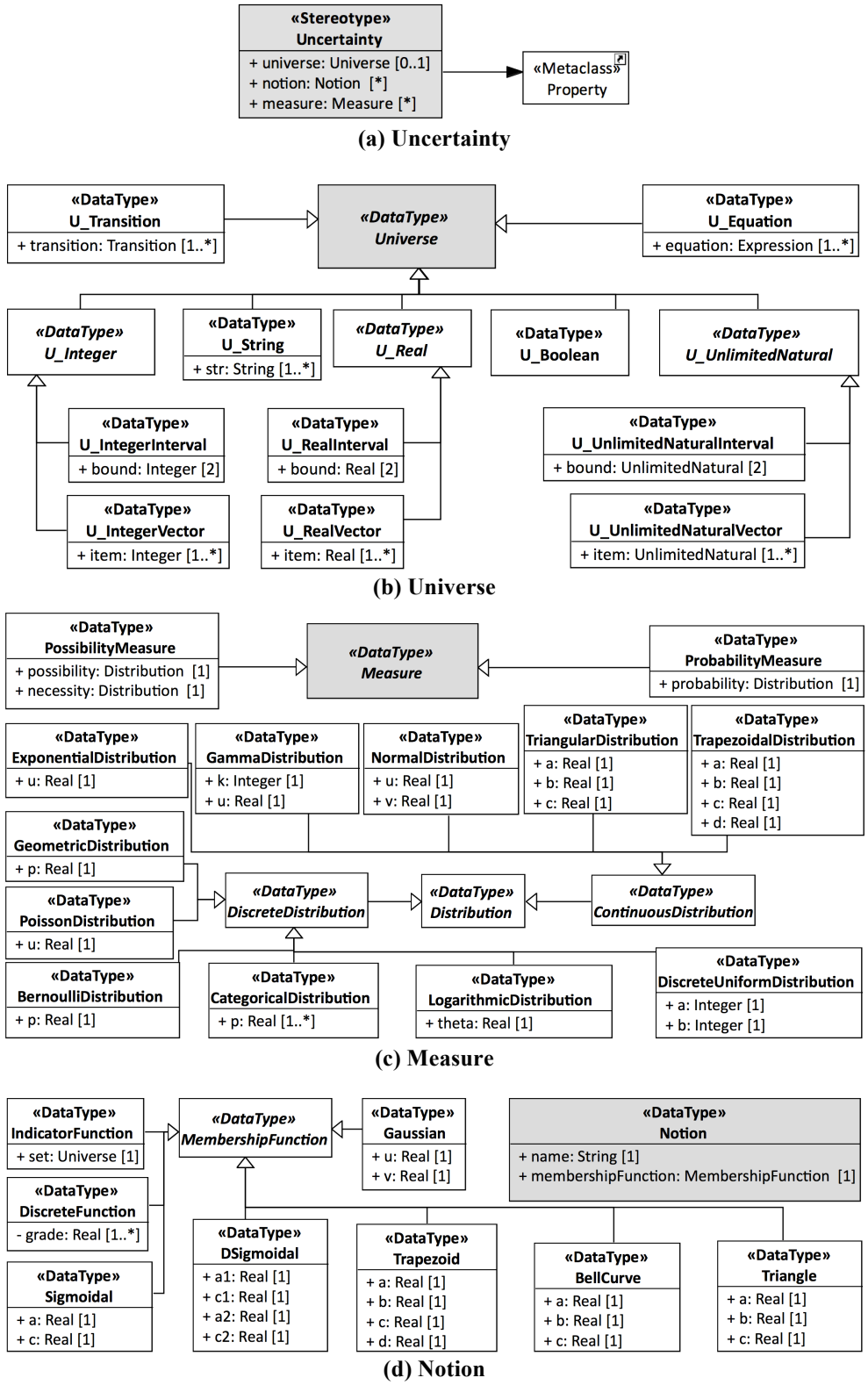

Figure 16 ShCPS Uncertainty Profile

For Level 3 uncertainty, the probability distribution is unknown. Only a rankable likelihood of each value is available. In this case, *PossibilityMeasure* can be used to specify the rankable likelihood, via possibility (C59) and necessity (C60) distributions, as shown in Figure 17

(Level 3). Based on MARTE [16], 11 types of *Distribution*s are defined in this profile (Figure 16 (c)). They can be used to specify both probability and possibility measures.

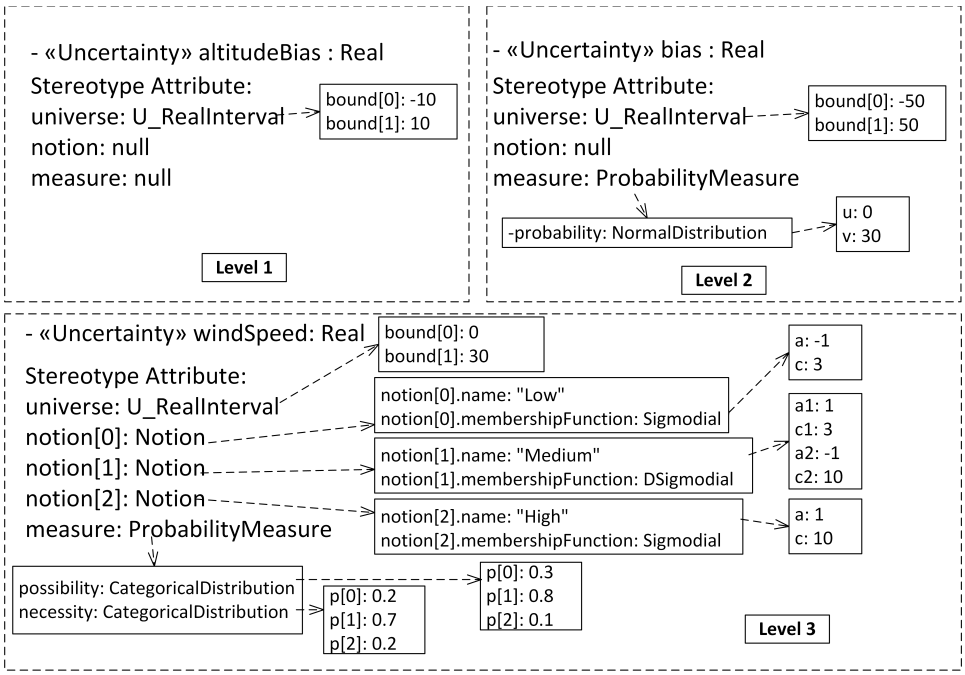

Figure 17 Specifications of Uncertainty

4.3.2 Model Uncertainty (A3)

The next step of building the ETRM is to specify uncertainties from sensors and actuators. Limited by the current state of knowledge, testers may not be able to determine every feature of each sensor and actuator. However, these features have effects on controllers' behaviors, such as *packetLossRate* of *MAVLink* and *altitudeBias* of *Barometer*. To support testing ShCPSs under uncertainties, uncertain features are defined as class attributes stereotyped with «Uncertainty». The uncertainty is quantified, by defining universes and notions of uncertain features, and by quantifying uncertainties with measures. The modeling process is summarized in Figure 18.

According to the type of an uncertain feature, one of the 13 types of *Universe* datatypes defined in ShCPS Uncertainty Profile (Section 4.3.1), can be chosen as the datatype of the uncertainty. For a numerical feature, if its value varies within a range, an interval datatype of universe (*U_IntegerInterval*, *U_RealInterval*, *U_UnlimitedNaturalInterval*) should be assigned to this uncertainty. Otherwise, a vector can be used to list all possible values. Depending on the level of uncertainty, the notions and measure are specified in the following way.

For Level 1 uncertainty, at a given point of time, the value of the uncertain feature is determined with a margin of error. They are specified via "universe" attribute of «Uncertainty», as shown in Figure 17 (Level 1). For the other two levels of uncertainty, prior to quantifying the likelihood of each value, the modeler should decide whether their

knowledge about the uncertain feature is qualitative or quantitative. If it is qualitative, a notion should be defined for each descriptive term, such as the three notions *Low*, *Medium*, and *High*, defined for uncertain feature *windSpeed* in Figure 17. Any *MembershipFunction* defined in the uncertainty profile (Section 4.3.1) can be used to define the elements of a notion.

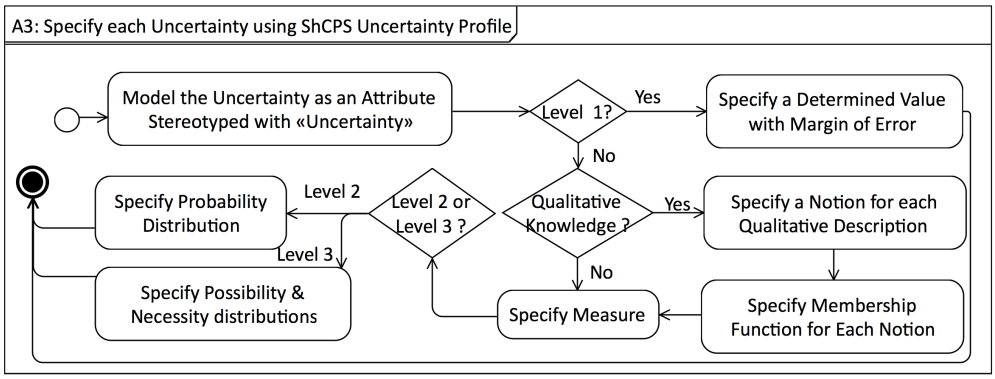

Figure 18 Model Uncertainty

After defining notions, modelers specify the measure of each uncertainty. For the Level 2 uncertainty, since the probability of each value is known, a *ProbabilityMeasure* is used to describe the likelihood. While *PossibilityMeasure* is adopted for the Level 3 uncertainty to state the rankable likelihood of each value. For both Level 2 and Level 3 uncertainties, appropriate probability, possibility, or necessity distributions can be chosen from the set of predefined *Distribution*s in the uncertainty profile (Section 4.3.1).

Since measures of Level 4 and Level 5 uncertainties are unknown, these uncertainties cannot be explicitly specified in the model. However, as testing proceeds, such uncertainties may transform to lower level uncertainties and be handled accordingly.

4.4 Model Testing Utilities with ShCPS Testing Profile

4.4.1 ShCPS Testing Profile

ShCPS Testing profile defines five stereotypes based on necessary concepts from the standard of Methods for Testing and Specification of Model-based Testing [39] (shown in Figure 19). «SystemUnderTest» denotes the testing target, i.e., ShCPS. «InputOperation» and «OutputOperation» extend *BehavioralFeature* representing testing interfaces used for controlling and monitoring the ShCPS. An «InputOperation» testing interface sends instructions to the «SystemUnderTest», whereas an «OutputOperation» testing interface queries state variable values. These types of operations facilitate test execution (to be discussed in Section 5). «FaultInjection» is a specialized «InputOperation» for faults injections to trigger self-healing behaviors. «TestStub» represents sensors, actuators, networks, and external systems, which are simulated/emulated by simulators/emulators. As explained in Section 1, self-healing behaviors are tested in a simulated environment. Hence, simulators or emulators play an important role in building and maintaining a realistic test execution

environment for ShCPSs. The "parameter" attribute of «TestStub» specifies configuration parameters required for launching stubs.

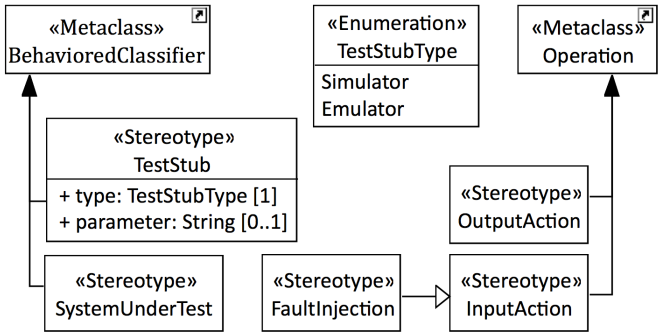

Figure 19 ShCPS Testing Profile

4.4.2 Model Test Utilities (A4)

The final step of the modeling is to bind the testing interfaces with the defined operations and to achieve the final ETRMs. Figure 20 presents the three stages of this step.

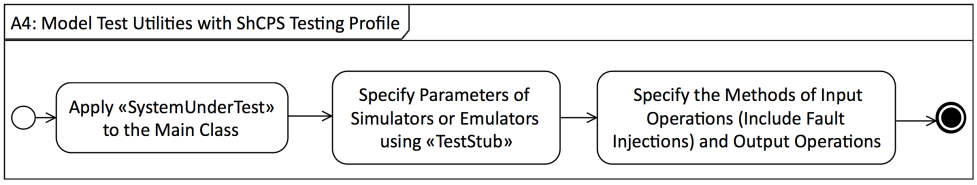

Figure 20 Model Test Utilities

First, the main class, which contains the entry point of the testing process, is stereotyped with «SystemUnderTest» to show the starting point of execution. Second, sensors, actuators, and physical processes, which are to be simulated or emulated, are annotated with «TestStub». The "parameter" attribute of «TestStub» captures configuration parameters of simulators or emulators. Figure 21 presents the parameters of *GPS*: *id* and *resolution*. They are all captured in the "parameter" attribute. By parsing this value, *TM-Executor* knows how to start and initialize a corresponding simulator to build the testing environment.

Third, operations defined in class diagrams can be stereotyped with «InputOperation», «OutputOperation» or «FaultInjection» to distinguish them from each other. For «InputOperation» and «FaultInjection», their input parameters capture input data of the corresponding testing interface. Such an operation is defined as an opaque behavior, which states the Uniform Resource Identifier (URI) of the corresponding testing interface, such as the method of *disableGPS()* shown in Figure 21. According to this URI, the testing interface *disableGPS()* will be invoked by *TM-Executor* whenever the operation is called.

For «OutputOperation», besides specifying the URI and parameters of the testing interface, the modeler should also associate each output parameter to a class attribute, so that the attribute can be updated by system's current state variable value obtained through the operation. To do so, the name of the output parameter is the same as the one of the corresponding attribute. In this way, each output parameter of an output operation is bound to

an attribute contained in the class owning this operation. For example in Figure 21, the output parameter of *getGPSPosition()*, is *position*, the same as attribute *position* of class *GPS* (Figure 11). Whenever the operation is invoked, attribute *position* is updated by the operation.

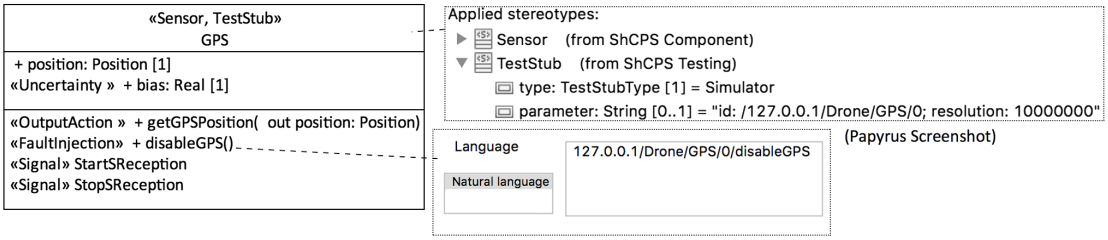

Figure 21 Test Utilities of GPS

5. The *TM-Executor* Framework

To enable EMBT for ShCPSs in the presence of environment uncertainties, we created the *MoSH* and *TM-Executor* frameworks. An overview of the frameworks including integrated tools is presented in Figure 22. The *MoSH* framework, i.e., UML profiles presented in Section 4, is implemented in Papyrus [40], a UML Modeling Tool. With *MoSH*, testers can develop the ETRM, which captures expected behaviors and environment uncertainties of the SUT. To execute the ETRM and test the SUT under environment uncertainties, we developed the *TM-Executor* framework as an extension to Moka [11], which is a Papyrus module for execution of UML models complying with fUML standard [10]. The key input for the *TM-Executor* framework is an ETRM created with *MoSH*.

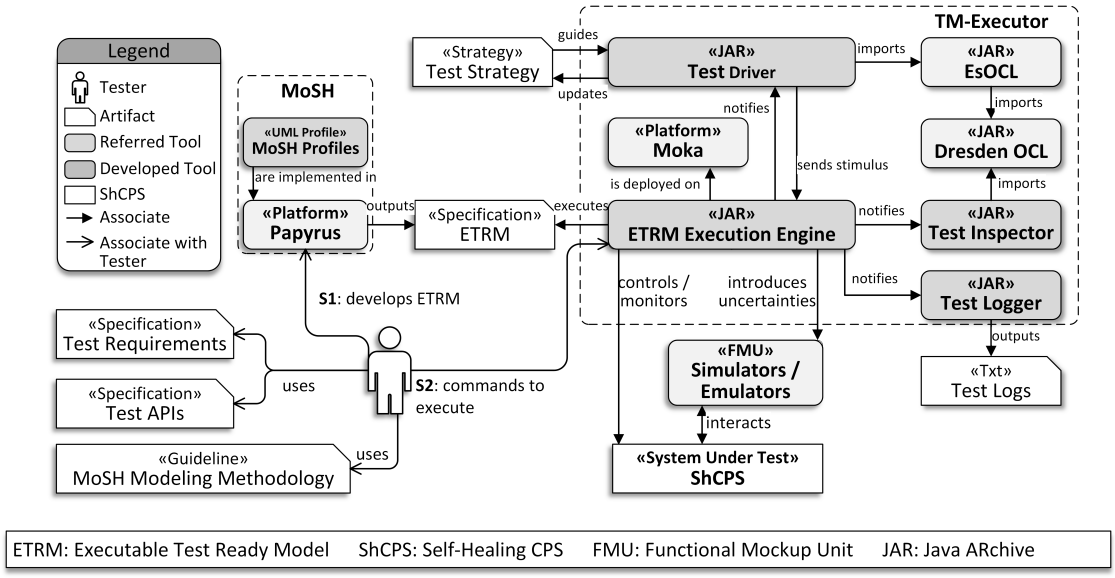

Figure 22 *TM-Executor* Framework

TM-Executor extends Moka in four ways. First, it extends Moka to execute ETRMs containing stereotypes from the *MoSH* profiles since Moka doesn't recognize stereotypes defined in the *MoSH* profiles. For example, as shown in Table 11, operations with stereotypes «InputOperation» and «OutputOperation» applied have specific semantics in our case and thus

we extended the existing semantics of the operations defined in fUML. Second, execution semantics for certain UML model elements are not yet defined in fUML, for instance, for *BroadcastSignalAction* and *ChangeEvent*. Thus, we defined their execution semantics ourselves and implemented them as extensions in Moka. Third, there exist defined semantics in fUML for certain UML metaclasses that do not serve our purpose and thus we extended them, for instance, for *State* as shown in Table 11. Appendix B presents the implementation of these extensions. Fourth, we implement test execution facilities (i.e., *Test Driver*, *Test Inspector*, and *Test Logger*) since Moka is a generic model execution engine and needed to be specialized for testing.

Table 11 Extensions to Moka Execution Semantics

Extension	UML Metaclass	Execution Model Element	Execution Semantic
Extensions for stereotypes	Operation stereotyped with «InputOperation»	InputOperation Execution	Invoke the test interface corresponding to the URI defined in the opaque behavior of the operation, taking input parameter values as inputs.
	Operation stereotyped with «OutputOperation»	OutputOperation Execution	Invoke the test interface corresponding to the URI defined in the opaque behavior of the operation. Update attributes values using the outputs of the test interface.
Extensions for additional metaclasses	BroadcastSignalAction	BroadcastSignal ActionActivation	Construct a signal using the values from argument pins and send the signal to all objects that are associated with the object from the source pin.
	ChangeEvent	ChangeEvent Occurrence	The change expression of change event is evaluated, whenever related attributes' values are updated. If the change expression becomes true, the change event occurs.
Extension to existing semantic	State	StateActivation	Besides the semantic defined in fUML, the state can be entered only if its state invariant is true.

While executing ETRMs, certain test data (i.e., input parameter values) are needed to trigger transitions with *CallEvents*. In ETRMs, the valid input parameter values are defined by transitions' guard specified as OCL constraints. To avoid a user manually providing valid values, we use an OCL constraint solver, i.e., EsOCL [8], which takes an OCL constraint as input and generates a set of values satisfying the constraint. During the execution of ETRMs, whenever test data is required to continue the execution of ETRMs, *ETRM Execution Engine* interacts with *Test Driver*, which invokes EsOCL with an OCL constraint. EsOCL solves the constraint and provides required test data to *Test Driver*, which subsequently supplies the test data to *ETRM Execution Engine* to continue the execution of ETRMs. Figure 23 presents an

example of this process. First, to execute ETRM, *ETRM Execution Engine* notifies *Test Driver* to trigger a transition. Second, as directed by a random testing strategy, *Test Driver* arbitrarily chooses the transition "*takeoff() [alt > 50 and alt < 200]*" and invokes EsOCL to generate an input value satisfying the guard condition "*[alt > 50 and alt < 200]*". Third, by solving the constraint, EsOCL returns a valid value for variable *alt*, i.e., *alt=100*, which is eventually used to invoke *takeoff()*.

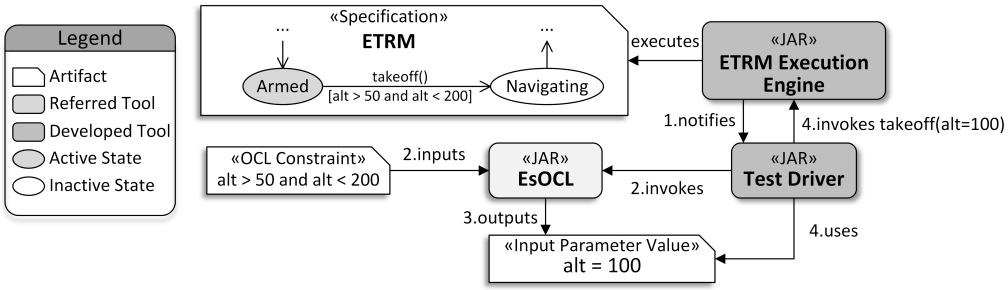

Figure 23 Example of Test Data Generation

During the execution of an ETRM, state invariants (OCL constraints) specified in the ETRM are evaluated to determine whether a fault is found. *Test Inspector* provides such functionality. *ETRM Execution Engine* invokes *Test Inspector* whenever the ETRM is updated upon the reception of new state variable values coming from testing interfaces. *Test Inspector* uses Dresden OCL [12] to evaluate a state invariant against actual values of the state variables. If the result of the evaluation is false, it means that there is a fault and the execution terminates; otherwise, there is no fault and the execution continues. Following the example shown in Figure 24, *Navigating* is the active state, and its state invariant is an OCL constraint defined on attribute *mode*. Whenever *mode* is updated by querying its value via a test interface, the *ETRM Execution Engine*, first, notifies *Test Inspector* to check system's actual state against the model. Second, *Test Inspector* invokes Dresden OCL to evaluate the state invariant based on the updated attribute value. Third, Dresden OCL returns "false", which means the system state is inconsistent with the active state in the ETRM. Thus, a fault is revealed, and *Test Inspector* terminates the execution.

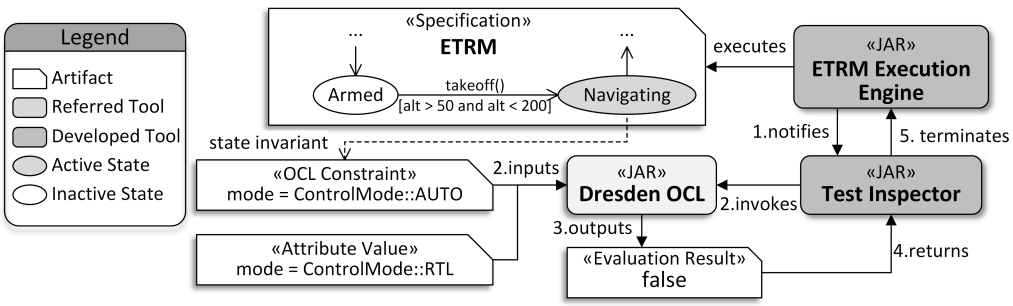

Figure 24 Example of Constraint Evaluation

Test Logger takes charge of creating test logs. Whenever a state machine in the ETRM changes its active state or an operation is invoked by *Test Driver*, *Test Logger* saves a log to record the change or stimulus. As a result, the logs keep the history of the execution process,

including the sequence of adaptations adopted for healing. In the case of a fault, i.e., an invariant violation detected by *Test Inspector*, the fault is logged as well. The logged fault, along with the history of execution can help to perform e.g., root cause analyses.

To support interactions among ETRM, SUT and simulators/emulators, we used the FMI standard [9], which is a tool independent standard to support model exchange and facilitate co-simulation of dynamic models. With FMI, *ETRM Execution Engine* executes ETRMs and in turn interacts with SUT. In addition, *ETRM Execution Engine* interacts with simulators/emulators using FMI to introduce uncertainties captured in ETRMs during their execution. Notice that our focus is only on testing the software of a self-healing CPS in the presence of environment uncertainties and this is the reason that hardware and its environment are simulated/emulated. Since the focus of testing is software, we captured its expected behavior as UML state machines. However, UML state machines cannot capture continuous behaviors and thus we used existing simulators/emulators. Supported by FMI, the input and output values of simulators' or emulators' interfaces can be modified to introduce uncertainties. The extent of each modification is determined by the uncertainty's universe, notions and measure defined in ETRMs (Section 4.3). Simulators and emulators can be developed using several modeling languages such as Modelica [41] and Simulink [42], which is however out of the scope of this paper.

6. Evaluation

This section presents the evaluation of *CMSU*, *MoSH*, and *TM-Executor*. Section 6.1 presents the experiment design. Experiment results are discussed in Section 6.2. We summrise the evaluation results in Section 6.3. In Section 6.4, we present threats to validity.

6.1 Experiment Design

As shown in Table 12, the experiment was designed to answer three research questions (RQ1-RQ3) through three carefully designed tasks (T_1-T_3). Experiment results were evaluated with a set of metrics and various numbers of case studies were involved in each task. In the rest of the section, we discuss the experiment design by following the research questions.

Table 12 Experiment Design

RQ	Task	Metrics	Case Studies
1	T_1: Mapping concepts and their relationships from a case study to the ones in *CMSU*	*Completeness*, *Correctness*	VCS, TMS, RFID-SC, DSRL, ISR, APR, RAMA, PeMS, VSS
2	T_2: Creating ETRMs with *MoSH*	*FunBeh, HealBeh, Diagnosis, Recovery, Uncertainty, TotalElem, StereoPer*	RAMA, PeMS, VSS
3	T_3: Testing a ShCPS with *TM-Executor* with a random strategy.	*TranTime, SynTime DataGenTime, EvalTime, UncIntrTime, DetectedFault*	RAMA

Table 13 Metrics and their Definitions

RQ	Metric	Description
1	*Completeness*	The number of ShCPS elements covered by *CMSU* (*Cov*) divided by the total number of elements in the case study (*Total*): *Cov* / *Total*
	Correctness	The number of relationships that are consistent with the ones in *CMSU* (*ConsAssoc*) divided by the total number of relationships in the case study (*TotalAssoc*): *ConsAssoc* / *TotalAssoc*
2	*FunBeh*	The number of functional behaviors in an ETRM
	HealBeh	The number of self-healing behaviors in an ETRM
	Diagnosis	The number of self-diagnosis behaviors in an ETRM
	Recovery	The number of self-recovery behaviors in an ETRM
	Uncertainty	The number of uncertainties captured in an ETRM
	TotalElem	The total number of model elements in an ETRM
	StereoPer	The number of stereotyped model elements in an ETRM (*SterElem*) divided by the total number of elements in the ETRM: *SterElem* / *TotalElem*
3	*TranTime*	Time for traversing a transition
	SynTime	Time for synchronizing ETRM, ShCPS, and simulators
	DataGenTime	Time for generating test data from a guard condition
	EvalTime	Time for evaluating a state invariant
	UncIntrTime	Time for introducing an uncertainty
	DetectedFault	The number of faults found by *TM-Executor* with the random strategy

RQ1: Is *CMSU* complete and correct to capture relevant concepts of the selected case studies?

With this research question, we first aimed to assess whether there are any concepts and/or relationships in the case studies that cannot be mapped to the concepts and/or relationships in *CMSU*. Doing so helps to find missing concepts in *CMSU* and consequently missing elements in the *MoSH*. Second, we aimed to know whether there are any relationships in the conceptual model of a case study that are mapped incorrectly to the relationships between the concepts in *CMSU*. Its purpose is to find any incorrect relationships in *CMSU*. We, therefore, defined the T_1 task (Table 12) to achieve these two objectives. We selected nine available ShCPS case studies to assess the quality of *CMSU*: Videoconferencing System (VCS) [43], Traffic Monitoring System (TMS) [44], Radio-frequency identification (RFID) supply chain (RFID-SC) [45], Distributed Systems Research Lab (DSRL) [46], Intelligent Service Robot (ISR) [47], Automatic Power Restoration System (APRS) [48], RAMA [13], freeway Performance Measurement System (PeMS) [49], and Video Streaming System (VSS) [50]. Experiment results were evaluated with the metrics of *Completeness* and *Correctness* (Table 12), which are defined in Table 13.

RQ2: Does *MoSH* provide a cost-effective way of creating ETRMs?

With this research questions, we first want to assess 1) how much additional modeling effort is required to create ETRMs for the selected case studies as compared to standard UML notations, and 2) the effectiveness of applying *MoSH* for modeling all identified self-healing behaviors and uncertainties of the selected case studies. For this RQ, we defined the T_2 task and defined a set of metrics (Table 12 and Table 13). Regarding case studies, we only used RAMA [13], PeMS [49], and (VSS) [50], due to the reason that the other six case studies used

to answer RQ1 do not provide detailed specifications of system structures, functional and self-healing behaviors.

RQ3: How is the performance of *TM-Executor* in terms of test execution?

TM-Executor performs test execution with model execution to test self-healing behaviors of ShCPSs in the presence of environment uncertainties. With this evaluation, we are interested in assessing how much time is required for *TM-Executor* to execute various test steps such as generating data from a guard condition and evaluating a state invariant. This gives us an indication whether *TM-Executor* is practically applicable in terms of its time performance. For RQ3, we defined the T_3 task, which involves testing a ShCPS in the presence of environment uncertainties as a proof-of-concept. We implemented a random test strategy, where during each test execution step, a random transition was selected for execution. During test execution, test data was generated using EsOCL, and uncertainties were simulated based on their universes, notions, and measures. Note that we do not aim to assess fault detection ability of test strategies, rather we aim to demonstrate, as a proof-of-concept, the feasibility of the complete EMBT solution. In the future, we plan to implement other test strategies. For RQ3, we chose to test RAMA [13], using the ETRM created to answer RQ2. We couldn't use PeMS and VSS for RQ3 as we didn't have access to the implementation to the ShCPS for testing.

6.2 Experiment Execution, Results, and Analyses

In this section, we provide details on experiment executions, results, and analyses, corresponding to each research question.

6.2.1 Results for RQ1

Based on the nine ShCPSs (Section 6.1), we evaluated and improved *CMSU*'s completeness and correctness, by following the steps summarized in Figure 25. Initially, we derived the conceptual model (*CMSU V.1*) from the existing literature on self-healing systems, CPSs, and uncertainty theories (Activity A1 in Figure 25). To evaluate its quality in terms of completeness and correctness, we abstracted ShCPS related concepts as well as concept relationships (*Cons. & Rels. from CSs. V.1*), from the nine ShCPSs' specifications (Activity A2.1 in Figure 25). *Cons. & Rels. from CSs. V.1* capture necessary entities required for defining expected self-healing behaviors and uncertainties of a ShCPS. For each abstracted concept or relationship, we tried to find a counterpart in *CMSU V.1* (Activity A2.2 in Figure 25). If the counterpart is missing, we further investigated whether the abstracted one is correctly identified. In case that it was correct, *CMSU V.1* was revised to cover the missing concept. In case that we identified a problem in the case studies, the wrong concept or relationship was corrected. After A2.2, we created a new version of abstracted concepts and relationships, i.e., *Cons. & Rels. from CSs. V.2*. At last, the refined conceptual model (*CMSU V.2*), generated by A2.2, was further refined by A3 via mapping from *Cons. & Rels. from CSs.*

V.2 to *CMSU V.2*. The final obtained *CMSU V.3* is presented in Section 3 and was implemented as UML profiles (presented in Section 4). In addition, Appendix A presents the statistics of the occurrence of each concept in the nine case studies for reference.

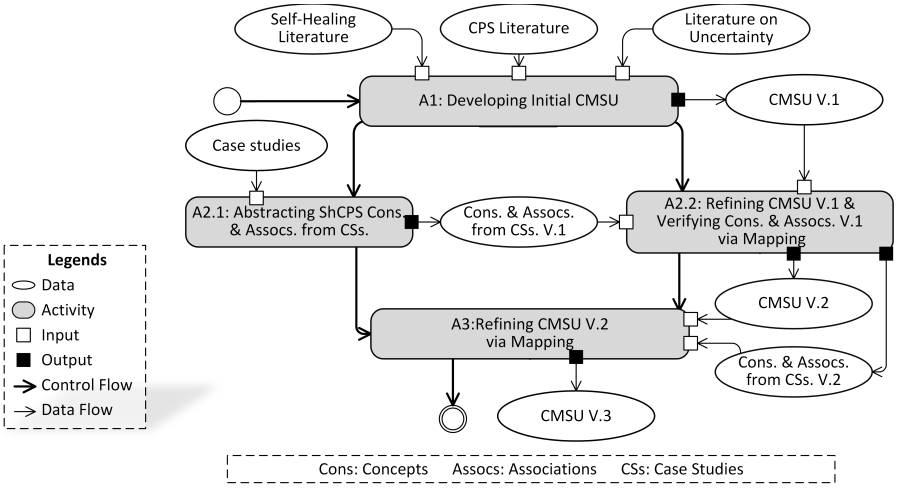

Figure 25 The Process of Developing *CMSU*

After the two-steps refinement, *CMSU* succeeded to correctly cover all the abstracted concepts and relationships and therefore the completeness, and correctness are 100% for all the nine case studies, which justifies that *CMSU* is complete and correct at least for the selected nine case studies.

6.2.2 *Results for RQ2*

To assess the additional effort required to applying *MoSH* for developing ETRMs, we report results for the *StereoPer* metric in Table 14. We needed to apply stereotypes from *MoSH* to 15%, 15%, and 19% of model elements for RAMA, PeMS, and VSS. On average, for the three case studies, we needed to apply stereotypes to 16% of model elements. This number gives us a rough indication of additional modeling effort required to use *MoSH* to create ETRMs.

Table 14 *MoSH* Evaluation Results (RQ2)

Metric	RAMA	PeMS	VSS	*Avg.*
TotalElem	377	144	97	206
FunBeh	10	7	3	7
HealBeh	4	5	2	4
Diagnosis	4	5	2	4
Recovery	9	5	3	6
Uncertainty	10	6	1	6
StereoPer	15%	15%	19%	16%

Regarding evaluating the effectiveness of *MoSH*, first, we provide statistics for various model elements in ETRMs for each case study in Table 15. Results in Table 15 give an indication of the complexity of ETRMs for the three case studies. Among the three case studies, RAMA is the most complicated one. It contains 10 functional behaviors and four self-healing behaviors. In total, 377 model elements were used to specify the 14 behaviors. In

contrast, PeMS and VSS are relatively simple, i.e., in total 144 model elements for specifying 12 behaviors of PeMS and 97 model elements for specifying five behaviors of VSS.

Table 15 Descriptive Statistics of the Model Elements (RQ2)

Element	RAMA	PeMS	VSS	*Avg.*
Class	10	7	4	7
Attribute	42	12	11	21
Operation	29	14	11	18
Signal	10	10	4	8
Association	9	7	3	6
State Machine	14	12	5	10
State	97	41	21	53
Transition	166	41	38	81
Total	377	144	97	204

Second, as discussed in Section 6.1, we collected statistics for the five metrics, two of which capture the number of functional and self-healing behaviors. As shown in Table 14 (the *FunBeh* and *HealBeh* rows) and Table 18 in Appendix A (the Functional Behavior and Self-Healing Behavior rows), all the identified functional and self-healing behaviors were captured in ETRMs. Moreover, self-diagnosis and self-recovery, the two key steps of self-healing behaviors, were also explicitly specified, as shown in the *Diagnosis* and *Recovery* rows in Table 14. They enable *TM-Executor* to rigorously test self-healing behaviors.

Table 16 Uncertainties in RAMA, PeMS and VSS (RQ2)

Case Study	Uncertainty	Level	Universe	Measure	Notions
RAMA	**Wind Direction**	3	0° ~ 360°	Possibility	Null
	Wind Velocity	3	0 ~ 30 m/s	Possibility	Low, Medium, High
	GPS Bias	2	-50 ~ 50 m	Probability	Null
	Motor Bias	2	-1 ~ 1 m/s^2	Probability	Null
	Barometer Altitude Bias	1	-10 ~ 10 m	N/A	N/A
	Barometer Climb Rate Bias	2	-0.5 ~ 0.5 m/s^2	Probability	Null
	Accelerometer Noise	2	-1 ~ 1 m/s^2	Probability	Null
	Gyro Noise	2	-0.1 ~ 0.1 rad/s	Probability	Null
	MAV Link Latency	2	0 ~ 600 ms	Probability	Null
	MAV Link Packet Loss Rate	2	0% ~ 5%	Probability	Null
PeMS	**Vehicle Speed**	3	10 ~ 120 km/h	Possibility	Null
	Vehicle Size	3	2000 ~ 5000 L	Possibility	Mini, Compact, Mid, Large
	Loop Detector Impedance	2	5 ~ 10 Ω	Probability	Null
	Loop Detector Voltage	2	3 ~ 4 V	Probability	Null
	Loop Detector Sensitivity	2	0.1 ~ 1 μH	Probability	Null
	Latency of ATM Link	2	0 ~ 1 s	Probability	Null
VSS	**Latency of Channel**	2	0 ~ 800 ms	Probability	Null

We specified 10 uncertainties for RAMA, six uncertainties for PeMS, and one uncertainty for VSS, as shown in Table 16. These uncertainties come from the environments of the software being tested and are related to sensors, actuators, and networks. Supported by *MoSH*, we could precisely define the universe, notions, and measure for each uncertainty. Based on

the uncertainty specifications, *TM-Executor* can introduce the uncertainties via simulators or emulators, which enables the testing of self-healing behaviors under uncertainties.

6.2.3 *Results for RQ3*

As discussed in Section 6.1, to answer RQ3, we used *TM-Executor* to test a real ShCPS (RAMA), based on the ETRM built in T_2 (Table 12). We investigated the efficiency of *TM-Executor* in terms of time taken for model execution (*TranTime* and *SynTime* in Table 13), test data generation (*DataGenTime* in Table 13), state invariant evaluation (*EvalTime* in Table 13), and uncertainty generation (*UncIntrTime* Table 13). As discussed in Section 6.1, we only implemented a random strategy for model execution and assessed if it can find faults (*DetectedFault* in Table 13).

We conducted the experiment on a single PC, with a processor Intel Core i7 2.6 GHz and 16 GB of RAM. As presented in Table 14, the ETRM developed for RAMA has 10 classes, four self-healing behaviors to handle four faults, and 14 state machines. We also have access to simulators for five sensors and one actuator. Ten uncertainties (Table 16) were introduced using these six simulators and we tested the four self-healing behaviors in the presence of these ten uncertainties. To reduce the effect of randomness of the executions, we repeated the experiment 10 times and report statistical values for each metric.

Table 17 summarizes the results. On average, traversing a transition took 5839 ms (i.e., 5.8 seconds), synchronization time took 1.5 ms, 95 ms for test data generation, and less than one millisecond for both state invariant evaluation and uncertainty generation. The maximum time taken for synchronization (34 ms), state invariant evaluation (1 ms), and uncertainty generation (<1 ms) are quite small. For test data generation, the maximum time was 104ms, where we used the EsOCL solver. Notice that depending on the complexity of a guard condition, time taken by test data generation may vary. In our case studies, in total there were 50 guards (OCL constraints), 27 of which contain six clauses. For the most complex constraint with six clauses, EsOCL took 104 ms to solve. During test execution, most of the time was spent on executing the simulators and the software of the ShCPS. The computational complexity of software and simulators determine the amount of time required to process a test stimulus. As shown in Table 17, after sending a stimulus to the ShCPS, its software maximally took 9421 ms (i.e., 9.4 seconds) to enter the target state. However, this time is not related to *TM-Executor*. Instead, it is system and simulator/emulator specific. In conclusion, *SynTime*, *DataGenTime*, *EvalTime*, and *UncIntrTime* are related to *TM-Executor*, which are all very small as shown in Table 17.

In the 10 runs of the experiment, one fault (state invariant violation) was detected by *TM-Executor* three times. This fault leads to the collision of the drone. Though a self-healing behavior helped the drone automatically avoid collisions with other vehicles, the drone failed to keep a safe distance from an intruding vehicle in the presence of uncertainties.

Table 17 *TM-Executor* Evaluation Result* (RQ3)

Metric	**Mean** (ms)	**Min.** (ms)	**Max.** (ms)
TranTime	5839	5270	9421
SynTime	1.5	1	34
DataGenTime	95	76	104
EvalTime	<1	<1	1
UncIntrTime	<1	<1	<1

*Min.: Minimum value, Max.: Maximum value

6.3 Overall Discussion

In this section, we provide an overall discussion based on the results presented in Section 6.2.1 to Section 6.2.3. As discussed in Section 6.2.1, we conclude that our conceptual model (*CMSU*) is complete and correct based on the results of evaluating with nine case studies. Such results give us an early indication on the completeness and correctness of *CMSU* and nonetheless more case studies are warranted. This conclusion is important as it forms the foundation for proposing the *MoSH* modeling methodology, another key contribution of the paper.

Based on the results presented in Section 6.2.2, we can conclude that, on average, we needed additional 16% of modeling effort to create ETRMs, which involved adding stereotypes to standard UML model elements. We understand that this is the simplest way of measuring modeling effort and more sophisticated ways to measure the modeling effort are required, e.g., conducting controlled experiments with human subjects (modelers) and assessing how much time is actually required by modelers when applying *MoSH* to develop ETRMs. In addition, we demonstrated the application of *MoSH* to create ETRMs for three diverse case studies of varying complexity. Such exercise gives us the evidence that *MoSH* is capable of modeling different ShCPSs to support testing in the presence of environment uncertainties. For the current evaluation, the first author of the paper created all the ETRMs. However, we acknowledge that a better evaluation would be to conduct a controlled experiment with more modelers with a diverse background to assess the applicability of *MoSH*. Conducting such controlled experiments (even in an academic setting) requires resources and opportunities, which are therefore expensive. We are however actively pursuing such opportunities.

Based on the results presented in Section 6.2.3, we conclude that the time taken by *TM-Executor* to perform various testing activities during test model execution was very small, i.e., in the order of milliseconds, with the exception of traversing a transition that, on average, took 5.8 seconds. Notice that this was the time taken by an SUT to execute the invoked operation/signal event on the transition, not by *TM-Executor* to invoke an event on the transition. Thus, such time is dependent on the implementation of an SUT and has nothing to do with the performance of *TM-Executor*.

6.4 Threats to Validity

Conclusion validity threats are related with factors that can affect conclusions drawn from experiment results. The random test strategy implemented in *TM-Executor* leads to different behaviors of the SUT being exercised. For different behaviors, the amount of time spent by *TM-Executor* to generate test data, evaluate constraints, etc., varies as well. Therefore, *TM-Executor*'s performance changes for each test execution. To deal with such a threat to conclusion validity, we repeated the experiment 10 times and applied statistics to analyze *TM-Executor*'s performance.

External validity threats concern the generalization of the results. One of the main *external validity threats* of the evaluation is that we only applied nine case studies to evaluate *CMSU* and applied three of them to evaluate *MoSH*. However, the nine case studies are from different domains and, for all the case studies, the evaluation results are consistent. Another *external validity threat* is that only one case study was employed to evaluate *TM-Executor*'s performance. However, 10 uncertainties and a number of state invariants, guards, operations, with various complexities, were exploited by *TM-Executor* to test self-healing behaviors under uncertainties. Nonetheless, additional case studies are needed to further generalize the results.

Construct validity threats refer to the degree to which the experiment setting (including two metrics for the *CMSU* evaluation, seven metrics for *MoSH* and six metrics for *TM-Executor* evaluation) reflects the construct under study (i.e., the quality of *CMSU*, the cost-effectiveness of *MoSH* and the performance of *TM-Executor*). To reduce the threats, we carefully selected and defined the metrics focusing on our overall objective of testing self-healing behaviors of ShCPSs under uncertainties. However, additional metrics and other ways of evaluation are also possible. For example, for the evaluation of *MoSH*, an alternate way to evaluate the modeling effort is to conduct a controlled experiment with real modelers and assess required effort. Nonetheless, conducting such experiments is very expensive in terms of time and resources required to execute such experiments. We are however looking for opportunities to conduct such controlled experiments in the future.

7. Related work

In this section, we discuss existing works related to *CMSU* in Section 7.1, *MoSH* in Section 7.2, and testing Self-Healing systems in Section 7.3. In Section 7.4, we summarize how our work advances the current state of the art.

7.1 Concepts of CPSs, Self-healing, and Uncertainty

After a decade's effort, key elements of ShCPSs have been identified by academic and industrial communities, which are adopted in our conceptual model *CMSU*. In [3], a CPS was defined as a set of heterogeneous physical units communicating via heterogeneous networks.

In this definition, physical units are recognized as the first-class objects of CPSs. Besides, the network was also considered important, as it enables communication among physical units. Our definition of CPSs is consistent with other definitions of CPSs: *"engineered systems that are built from, and depend upon, the seamless integration of computational and physical components"* [51] and *"a set of physical systems controlled in a principled manner via engineering technologies"* [52].

Sensors and actuator are captured as interfaces between computational and physical components in [53]. CPSs are characterized by integrating computation and physical processes [54] and the primary goal of a CPS is to efficiently control physical processes [55]. The authors of [5] identified detection, diagnosis, and recovery as the three main steps of self-healing. In addition, three types of recovery policies were explained and evaluated in [25, 56].

Though CPS and self-healing system have been the focus of research for years, only a few studies tried to associate them together. *CMSU* aims to build a common understanding of ShCPSs by capturing key elements from both CPSs and self-healing communities. Besides the key elements defined in the literature, we identified a few new concepts. First, we identified *Situation* to represent inherent uncertain events in the operational environment of a ShCPS, which is though studied, e.g., in [57, 58], for other purposes and in different contexts. Second, we adopted the definitions of fault and error from [19]. Third, adaptation actions, the classification of probes and effectors were elicited from fault diagnosis and recovery process. Fourth, inspired by goal oriented self-healing approaches [59], goals of self-healing behaviors were captured in *MoSH* as well.

How to cope with uncertainty is a grand challenge and a definition and taxonomy of uncertainty in the context of CPSs is difficult to find [60]. In the past, the effort was mostly spent on identifying uncertainty sources in self-healing CPSs. The authors of [6] proposed a taxonomy of uncertainty sources in dynamically adaptive systems at the requirement, design, and execution phases along with existing mitigation techniques for each type of uncertainties. The taxonomy is generic and therefore not designed for a specific usage and needs to be specialized for specific applications. In [61], the authors gave another nine uncertainty sources in self-adaptive systems, which need to be considered during design. We, however, build an uncertainty conceptual model from the perspective of testing. We extend the definition of uncertainty provided in [7] and define the uncertainty as: *"the lack of knowledge about the value of an uncertain feature at a given point of time during a testing process"* (Section 3.3). The uncertain feature, along with its universe and notions, and the measure of uncertainty are defined to more rigorously quantify the current state of knowledge.

In summary, despite numerous approaches proposed [2, 5, 62], a conceptual model of CPSs and their self-healing behaviors together with uncertainty is still missing. We, in the paper, took the initiative and constructed such a conceptual model, aiming at providing a common

ground for understanding ShCPSs under uncertainty and facilitating analyses in the future. However, we believe that this conceptual model is an initial attempt and must be specialized such as for other types of autonomic behaviors, e.g., self-configuring and for different types of analyses such as model-based testing.

7.2 UML-based CPS and Self-healing related Modeling

To tackle the intrinsic complexity of CPSs, researcher proposed to adopt model-based engineering [63], which uses models to facilitate system design, development, verification, and validation. Since a CPS is an integration of computation and physical processes, it is typically modeled as a hybrid system where physical processes are specified as continuous-time models and computation parts are defined as discrete models [64]. For physical processes, several modeling tools are ready to be used to specify continuous-time models, such as Simulink [42], OpenModelica [41], SystemC [65], and Ptolemy II [66]. Regarding the computational part, UML is the most broadly used modeling language. With the help of the FMI standard [9], heterogeneous models can be executed together.

Using UML's profiling mechanism, several extensions of UML have been developed, e.g., Systems Modeling Language (SysML) [67], Modeling and Analysis of Real-time Embedded Systems (MARTE) [16], Dependability Analysis Modeling (DAM) [68], and UML Profile for Modeling Quality Of Service And Fault Tolerance Characteristics And Mechanisms (QFTP) [69]. Though these UML profiles extend UML's capability to model complex systems, they are either too general or too limited to be used to build ETRMs for testing self-healing behaviors of ShCPSs under uncertainty. While SysML and MARTE provide useful modeling constructs to specify continuous system dynamics and non-functional properties, they do not provide sufficient modeling elements to precisely capture expected self-healing behaviors and uncertainties. DAM and QFTP can be used to support the modeling of fault tolerance mechanisms. Since self-healing behaviors can be realized by runtime adaptation rather than fault tolerance, they are not adequate for developing ETRMs.

To explicitly capture self-healing behaviors of ShCPSs in the presence of uncertainties, we propose a modeling framework, *MoSH*, in this paper. It provides four UML profiles and a modeling methodology to capture the test configuration of the ShCPS under test (including system structure, functional behaviors, self-healing behaviors, uncertainties and testing interfaces). This is not covered by existing works in the literature.

7.3 Testing Self-healing Systems

Fault injection is a straightforward method to test recovery mechanisms of self-healing systems. By introducing faults, self-healing behaviors can be exercised; Otherwise, they will rarely be triggered. Normally, it requires a fault model to capture potential faults. In [45], a fault model is built based on five types of faults (i.e., application hang, component crash, stale

service, denial of service and excessive thread allocation). According to the model, faults are introduced by deploying and activating faulty Java Beans on a platform, which leads to the execution of self-healing behaviors. While in [70] the authors propose to use context models to simulate sensor data under different adverse conditions. Taking simulated data as system input, self-healing behaviors can be evaluated. Similarly, in [71] a simulated architecture model is used as input to test the feedback loop of a self-healing system. This enables the self-healing mechanism to be assessed at the early phase of the system development lifecycle.

The main issue of these testing approaches is that they only concern self-healing behaviors, with functional behaviors ignored. Thus, the strategy that determines when to introduce faults is missing. Moreover, uncertainties are not considered either. Alternatively, in *MoSH*, both functional and self-healing behaviors are captured, together with system structure, uncertainties, and testing interfaces. Via change events with «Fault», fault injection points can be explicitly specified as well.

Besides traditional offline testing methods that generate test cases before test execution, adaptive testing is identified as a crucial technique to validate runtime adaptations performed by self-healing systems [1]. An adaptive testing framework, named as Proteus, was proposed in [72]. Proteus takes a set of test cases as input. Due to runtime adaptations, initial test cases may become invalid. To keep them valid, Proteus uses an evolutionary algorithm to adapt the test cases at runtime, based on the heuristics of false positive and false negative. Although this method enables adaptive testing, the fault detection ability of this method highly depends on the quality of the initial test cases. Achieving a qualified set of test cases is still a challenge. The same issue exists for another adaptive testing framework, presented in [73]. Whenever a runtime adaption happens, the framework determines the affected components by dependency analysis. Accordingly, a minimal set of test cases is executed to check the correctness of the adaptation.

Different from the two approaches, we use ETRMs to guide the testing process. Since ETRMs can be executed together with the system under test, there is no need to generate test cases. Alternatively, based on the runtime information provided by ETRMs, test data is dynamically generated to follow a test path, which is also determined at runtime. As a result, ETRMs enables the adaptive testing of ShCPSs in the presence of uncertainties.

7.4 Summary

In conclusion, our work advances the current state of the art in several ways. First, even though some existing works define self-healing concepts [2, 5] and uncertainty related concepts [6, 7], there is no a single work that defines concepts of self-healing and uncertainty together in the context of CPSs. To this end, *CMSU* is a comprehensive conceptual model that builds on the literature [3, 19, 25, 51-56] to conceptualize self-healing behaviors of CPSs and

uncertainty together. Second, even though there exist several modeling notations that can be used to model self-healing behaviors such as [68, 69]; however, none of them provide a complete executable test modeling solution to create ETRMs for testing self-healing behaviors in the presence of environment uncertainties. *MoSH* provides a complete integrated modeling solution based on existing standards to provide such support. Note that *MoSH* was developed exclusively for test modeling. Such type of modeling is only concerned with modeling test interfaces (e.g., test APIs to send a stimulus to the SUT and modeling test data specifications), and expected behaviors of a ShCPS in the presence of uncertainties. Finally, there exist some adaptive test strategies [72, 73] to test self-healing behaviors; however, there is no evidence that such strategies can be adopted to perform testing in the presence of environment uncertainties. Though we didn't define new test strategies in this work, we implement *TM-Executor* to test self-healing behaviors of ShCPSs in the presence of uncertainty. Such a framework can integrate adaptive test strategies and we plan to devise such test strategies as part of our future work.

8. Conclusion and Future Work

Self-Healing Cyber-Physical Systems (ShCPSs) have the built-in capability to diagnose faults and recover from these faults at the runtime by themselves. Such systems operate in a highly unpredictable environment leading to uncertainty in their behaviors and thus these systems must deal with such uncertainty even during the process of fault recovering. Towards the direction of proposing an Executable Model-Based Testing (EMBT) approach to test the self-healing behaviors of ShCPSs in the presence of environment uncertainties, in the paper, we proposed an executable test modeling framework (*MoSH*) and a test model execution framework (*TM-Executor*). *MoSH* and *TM-Executor* were evaluated with several case studies and the feasibility of the EMBT solution was demonstrated by applying *TM-Executor* to test a real-world case study, with a random test strategy implemented as a proof-of-concept. In the future, we plan to implement more advanced strategies for test model execution test data generation, and uncertainty introduction.

9. References

[1]. De Lemos, R., Giese, H., Müller, H.A., Shaw, M., Andersson, J., Litoiu, M., Schmerl, B., Tamura, G., Villegas, N.M., Vogel, T.: Software engineering for self-adaptive systems: A second research roadmap. Software Engineering for Self-Adaptive Systems II, pp. 1-32. Springer (2013)

[2]. Ghosh, D., Sharman, R., Rao, H.R., Upadhyaya, S.: Self-healing systems—survey and synthesis. Decision Support Systems, vol.42, pp.2164-2185 (2007)

[3]. Zhang, M., Selic, B., Ali, S., Yue, T., Okariz, O., Norgren, R.: Understanding Uncertainty in Cyber-Physical Systems: A Conceptual Model. In: Modelling Foundations and Applications: 12th European Conference, ECMFA (2015)

[4]. Utting, M., Pretschner, A., Legeard, B.: A taxonomy of model-based testing. (2006)

[5]. Psaier, H., Dustdar, S.: A survey on self-healing systems: approaches and systems. Computing, vol.91, pp.43-73 (2011)
[6]. Ramirez, A.J., Jensen, A.C., Cheng, B.H.: A taxonomy of uncertainty for dynamically adaptive systems. In: Software Engineering for Adaptive and Self-Managing Systems (SEAMS), 2012 ICSE Workshop on, pp. 99-108 (2012)
[7]. Walker, W.E., Lempert, R.J., Kwakkel, J.H.: Deep uncertainty. Encyclopedia of operations research and management science, pp. 395-402. Springer (2013)
[8]. Ali, S., Iqbal, M.Z., Arcuri, A., Briand, L.C.: Generating test data from OCL constraints with search techniques. IEEE Transactions on software engineering, vol.39, pp.1376-1402 (2013)
[9]. Blochwitz, T., Otter, M., Akesson, J., Arnold, M., Clauss, C., Elmqvist, H., Friedrich, M., Junghanns, A., Mauss, J., Neumerkel, D.: Functional mockup interface 2.0: The standard for tool independent exchange of simulation models. In: Proceedings of the 9th International MODELICA Conference, pp. 173-184 (2012)
[10]. OMG: Semantics Of A Foundational Subset For Executable UML Models V1.2.1. formal/2016-01-05 (2016)
[11]. Tatibouet, J.: Moka – A simulation platform for Papyrus based on OMG specifications for executable UML. In: EclipseCon, (2016)
[12]. Demuth, B., Wilke, C.: Model and object verification by using Dresden OCL. In: Proceedings of the Russian-German Workshop Innovation Information Technologies: Theory and Practice, Ufa, Russia, pp. 687-690 (2009)
[13]. Holub, O., Hanzálek, Z.: Low-cost reconfigurable control system for small UAVs. IEEE Transactions on Industrial Electronics, vol.58, pp.880-889 (2011)
[14]. Selic, B.: A systematic approach to domain-specific language design using UML. In: Object and Component-Oriented Real-Time Distributed Computing, 2007. ISORC'07. 10th IEEE International Symposium on, pp. 2-9 (2007)
[15]. Kandé, M.M., Strohmeier, A.: Towards a UML profile for software architecture descriptions. In: International Conference on the Unified Modeling Language, pp. 513-527 (2000)
[16]. OMG: Profile for modeling and analysis of real-time and embedded systems (MARTE). formal/2011-06-02 (2011)
[17]. Yau, S.S., Wang, Y., Huang, D., In, H.P.: Situation-aware contract specification language for middleware for ubiquitous computing. In: Distributed Computing Systems, 2003. FTDCS 2003. Proceedings. The Ninth IEEE Workshop on Future Trends of, pp. 93-99 (2003)
[18]. Lee, E.A.: Cyber physical systems: Design challenges. In: 2008 11th IEEE International Symposium on Object and Component-Oriented Real-Time Distributed Computing (ISORC), pp. 363-369 (2008)
[19]. Avižienis, A., Laprie, J.-C., Randell, B., Landwehr, C.: Basic concepts and taxonomy of dependable and secure computing. Dependable and Secure Computing, IEEE Transactions on, vol.1, pp.11-33 (2004)
[20]. McKinley, P.K., Sadjadi, S.M., Kasten, E.P., Cheng, B.H.: Composing adaptive software. Computer, vol.37, pp.56-64 (2004)
[21]. Dardenne, A., Van Lamsweerde, A., Fickas, S.: Goal-directed requirements acquisition. Science of Computer Programming, vol.20, pp.3-50 (1993)
[22]. Koskimies, K., Mäkinen, E.: Automatic synthesis of state machines from trace diagrams. Software: Practice and Experience, vol.24, pp.643-658 (1994)
[23]. Garlan, D., Cheng, S.-W., Huang, A.-C., Schmerl, B., Steenkiste, P.: Rainbow: Architecture-based self-adaptation with reusable infrastructure. Computer, vol.37, pp.46-54 (2004)
[24]. Association, I.S.: Systems and software engineering—Vocabulary ISO/IEC/IEEE 24765: 2010. Iso/Iec/Ieee 24765 (2010)
[25]. Kephart, J.O., Walsh, W.E.: An artificial intelligence perspective on autonomic computing policies. In: Policies for Distributed Systems and Networks, 2004. POLICY 2004. Proceedings. Fifth IEEE International Workshop on, pp. 3-12 (2004)

[26]. Subramanian, N., Chung, L.: Software architecture adaptability: an NFR approach. In: Proceedings of the 4th International Workshop on Principles of Software Evolution, pp. 52-61 (2001)
[27]. Blanke, M., Schröder, J.: Diagnosis and fault-tolerant control. Springer (2006)
[28]. Venkatasubramanian, V., Rengaswamy, R., Yin, K., Kavuri, S.N.: A review of process fault detection and diagnosis: Part I: Quantitative model-based methods. Computers & chemical engineering, vol.27, pp.293-311 (2003)
[29]. Siripongwutikorn, P., Banerjee, S., Tipper, D.: A survey of adaptive bandwidth control algorithms. Communications Surveys & Tutorials, IEEE, vol.5, pp.14-26 (2003)
[30]. Garlan, D., Schmerl, B.: Model-based adaptation for self-healing systems. In: Proceedings of the first workshop on Self-healing systems, pp. 27-32 (2002)
[31]. Koutsoumpas, V.: A model-based approach for the specification of a virtual power plant operating in open context. In: Proceedings of the First International Workshop on Software Engineering for Smart Cyber-Physical Systems, pp. 26-32 (2015)
[32]. Simmonds, J., Ben-David, S., Chechik, M.: Monitoring and recovery of web service applications. The smart internet, pp. 250-288. Springer (2010)
[33]. Cheng, S.-W., Garlan, D., Schmerl, B.: Architecture-based self-adaptation in the presence of multiple objectives. In: Proceedings of the 2006 international workshop on Self-adaptation and self-managing systems, pp. 2-8 (2006)
[34]. Kallenberg, O.: Foundations of modern probability. Springer Science & Business Media (2006)
[35]. Dubois, D., Prade, H.: Possibility theory: an approach to computerized processing of uncertainty. Springer Science & Business Media (2012)
[36]. OMG: Unified Modeling Language V2.5. formal/15-03-01 (2015)
[37]. (OMG), O.M.G.: Concrete Syntax For A UML Action Language: Action Language For Foundational UML (ALF). (2013)
[38]. Sivanandam, S., Sumathi, S., Deepa, S.: Introduction to fuzzy logic using MATLAB. Springer (2007)
[39]. ETSI: Methods for Testing and Specification (MTS); Model-Based Testing (MBT); Requirements for Modelling Notations, V1.1.1. ETSI ES 202 951 (2011)
[40]. Lanusse, A., Tanguy, Y., Espinoza, H., Mraidha, C., Gerard, S., Tessier, P., Schnekenburger, R., Dubois, H., Terrier, F.: Papyrus UML: an open source toolset for MDA. In: Proc. of the Fifth European Conference on Model-Driven Architecture Foundations and Applications (ECMDA-FA 2009), pp. 1-4 (2009)
[41]. Fritzson, P., Aronsson, P., Pop, A., Lundvall, H., Nystrom, K., Saldamli, L., Broman, D., Sandholm, A.: OpenModelica-A free open-source environment for system modeling, simulation, and teaching. In: IEEE International Symposium on Computer-Aided Control Systems Design, pp. 1588-1595 (2006)
[42]. Dabney, J.B., Harman, T.L.: Mastering simulink. Pearson/Prentice Hall (2004)
[43]. Ali, S., Briand, L.C., Hemmati, H.: Modeling robustness behavior using aspect-oriented modeling to support robustness testing of industrial systems. Software & Systems Modeling, vol.11, pp.633-670 (2012)
[44]. Vromant, P., Weyns, D., Malek, S., Andersson, J.: On interacting control loops in self-adaptive systems. In: Proceedings of the 6th International Symposium on Software Engineering for Adaptive and Self-Managing Systems, pp. 202-207 (2011)
[45]. Gama, K., Donsez, D.: Deployment and activation of faulty components at runtime for testing self-recovery mechanisms. ACM SIGAPP Applied Computing Review, vol.14, pp.44-54 (2014)
[46]. Cioara, T., Anghel, I., Salomie, I., Dinsoreanu, M., Copil, G., Moldovan, D.: A reinforcement learning based self-healing algorithm for managing context adaptation. In: Proceedings of the 12th International Conference on Information Integration and Web-based Applications & Services, pp. 859-862 (2010)
[47]. Park, J., Lee, S., Yoon, T., Kim, J.M.: An autonomic control system for high-reliable CPS. Cluster Computing, vol.18, pp.587-598 (2015)

[48]. Staszesky, D., Craig, D., Befus, C.: Advanced feeder automation is here. IEEE Power and Energy Magazine, vol.3, pp.56-63 (2005)
[49]. Lu, X.-Y., Varaiya, P., Horowitz, R., Palen, J.: Faulty loop data analysis/correction and loop fault detection. In: 15th World Congress on Intelligent Transport Systems and ITS America's 2008 Annual Meeting, (2008)
[50]. Ryu, B.-H., Jeon, D., Kim, D.-H.: A Robust Video Streaming Based on Primary-Shadow Fault-Tolerance Mechanism. In: International Conference on Ubiquitous Computing and Multimedia Applications, pp. 66-75 (2011)
[51]. NSF: Cyber Physical Systems. NSF 14-542 (2014)
[52]. Kim, K.-D., Kumar, P.R.: Cyber–physical systems: A perspective at the centennial. Proceedings of the IEEE, vol.100, pp.1287-1308 (2012)
[53]. Lee, E.A., Seshia, S.A.: Introduction to embedded systems: A cyber-physical systems approach. Lee & Seshia (2011)
[54]. Shi, J., Wan, J., Yan, H., Suo, H.: A survey of cyber-physical systems. In: Wireless Communications and Signal Processing (WCSP), 2011 International Conference on, pp. 1-6 (2011)
[55]. Sridhar, S., Hahn, A., Govindarasu, M.: Cyber–physical system security for the electric power grid. Proceedings of the IEEE, vol.100, pp.210-224 (2012)
[56]. White, S.R., Hanson, J.E., Whalley, I., Chess, D.M., Kephart, J.O.: An architectural approach to autonomic computing. In: null, pp. 2-9 (2004)
[57]. Bujorianu, M.C., Bujorianu, M.L., Barringer, H.: A unifying specification logic for cyber-physical systems. In: Control and Automation, 2009. MED'09. 17th Mediterranean Conference on, pp. 1166-1171 (2009)
[58]. Moreno, G.A., Cámara, J., Garlan, D., Schmerl, B.: Proactive Self-Adaptation under Uncertainty: a Probabilistic Model Checking Approach. In: Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering, pp. 1-12 (2015)
[59]. Morandini, M., Penserini, L., Perini, A.: Automated mapping from goal models to self-adaptive systems. In: Proceedings of the 2008 23rd IEEE/ACM International Conference on Automated Software Engineering, pp. 485-486 (2008)
[60]. Bures, T., Weyns, D., Berger, C., Biffl, S., Daun, M., Gabor, T., Garlan, D., Gerostathopoulos, I., Julien, C., Krikava, F.: Software Engineering for Smart Cyber-Physical Systems--Towards a Research Agenda: Report on the First International Workshop on Software Engineering for Smart CPS. ACM SIGSOFT Software Engineering Notes, vol.40, pp.28-32 (2015)
[61]. Esfahani, N., Malek, S.: Uncertainty in self-adaptive software systems. Software Engineering for Self-Adaptive Systems II, pp. 214-238. Springer (2013)
[62]. Khaitan, S.K., McCalley, J.D.: Design techniques and applications of cyberphysical systems: a survey. IEEE Systems Journal, vol.9, pp.350-365 (2014)
[63]. Ramos, A.L., Ferreira, J.V., Barceló, J.: Model-based systems engineering: An emerging approach for modern systems. IEEE Transactions on Systems, Man, and Cybernetics, Part C (Applications and Reviews), vol.42, pp.101-111 (2012)
[64]. Derler, P., Lee, E., Vincentelli, A.S.: Modeling cyber–physical systems. Proceedings of the IEEE, vol.100, pp.13-28 (2012)
[65]. Black, D.C., Donovan, J., Bunton, B., Keist, A.: SystemC: From the ground up. Springer Science & Business Media (2011)
[66]. Ptolemaeus, C.: System design, modeling, and simulation: using Ptolemy II. Ptolemy. org Berkeley (2014)
[67]. Friedenthal, S., Moore, A., Steiner, R.: A practical guide to SysML: the systems modeling language. Morgan Kaufmann (2014)
[68]. Bernardi, S., Merseguer, J., Petriu, D.C.: A dependability profile within MARTE. Software & Systems Modeling, vol.10, pp.313-336 (2011)
[69]. OMG: Profile for modeling quality of service and fault tolerance characteristics and mechanisms. formal/2008-04-05 (2008)

[70]. Huebscher, M.C., McCann, J.A.: Simulation model for self-adaptive applications in pervasive computing. In: Database and Expert Systems Applications, 2004. Proceedings. 15th International Workshop on, pp. 694-698 (2004)
[71]. Hänsel, J., Vogel, T., Giese, H.: A Testing Scheme for Self-Adaptive Software Systems with Architectural Runtime Models. In: Self-Adaptive and Self-Organizing Systems Workshops (SASOW), 2015 IEEE International Conference on, pp. 134-139 (2015)
[72]. Fredericks, E.M., Cheng, B.H.: Automated generation of adaptive test plans for self-adaptive systems. In: Appear in Proceedings of 10th International Symposium on Software Engineering for Adaptive and Self-Managing Systems. SEAMS, pp. 157-168 (2015)
[73]. Lahami, M., Krichen, M., Jmaiel, M.: Safe and efficient runtime testing framework applied in dynamic and distributed systems. Science of Computer Programming, vol.122, pp.1-28 (2016)

Appendix A Descriptive Statistics of the Case Studies

We selected nine ShCPS case studies to evaluate the completeness and correctness of *CMSU*. Table 18 and Table 19 show statistic of each concept's occurrence, abstracted from the nine case studies

Table 18 Descriptive Statistics of the Case Studies

Concept	VCS	TMS	APRS	RFID-SC	DSRL	ISR	RAMA	PeMS	VSS
Self-Healing CPS	1	1	1	1	1	1	1	1	1
PhysicalProcess	2	1	1	1	2	1	1	1	1
Network	1	1	1	1	1	1	1	1	1
PhysicalUnit	7	5	5	4	8	2	2	4	2
Sensor	3	1	3	1	4	5	5	2	1
Controller	7	2	2	4	9	2	3	4	2
Actuator	2	0	5	0	4	1	1	0	0
Functional Behavior	4	1	1	1	4	1	10	7	3
Self-Healing Behavior	2	1	2	4	4	5	4	5	2
Goal	1	1	1	1	1	1	1	1	1
State	21	6	22	11	32	36	97	41	21
Probe	2	1	3	5	4	5	5	2	2
Effector	2	1	5	1	4	2	1	2	1
Measurement	2	1	5	5	7	4	4	5	2
Self-Diagnosis	2	1	5	4	4	5	4	5	2
Self-Recovery	4	3	3	4	4	5	9	5	3
Fault	2	1	3	4	4	5	4	5	2
Error	2	1	5	4	4	5	4	5	2
RecoveryPolicy	4	3	3	4	1	5	9	5	3
AdaptationAction	4	3	10	3	8	4	4	2	2
Uncertainty	3	1	4	4	5	6	10	6	1
Total	78	36	90	67	115	102	180	109	55

Table 19 Descriptive Statistics of Categories of Probe, RecoveryPolicy, Effector, and Uncertainty

Concept		VCS	TMS	APRS	RFID-SC	DSRL	ISR	RAMA	PeMS	VSS	*P*
Probe	**PerformanceProbe**	2	0	0	3	0	0	0	2	2	31%
	EventProbe	0	1	0	2	0	0	0	0	0	10%
	PhysicalProcessProbe	0	0	3	0	4	5	5	0	0	59%
Recovery Policy	**ActionPolicy**	4	3	2	4	0	5	9	5	3	94%
	GoalPolicy	0	0	1	0	0	0	0	0	0	3%
	UtilityFunctionPolicy	0	0	0	0	1	0	0	0	0	3%
Effector	**ParameterEffector**	2	0	0	0	0	0	0	0	0	11%
	ArchitectureEffector	0	1	0	1	0	0	0	0	1	16%
	ControlEffector	0	0	5	0	4	2	1	2	0	73%
Uncertainty	**Level 1**	0	0	0	2	0	0	0	0	0	5%
	Level 2	3	1	3	2	5	6	8	6	1	87%
	Level 3	0	0	1	0	0	0	2	0	0	8%
	Level 4	0	0	0	0	0	0	0	0	0	0%
	Level 5	0	0	0	0	0	0	0	0	0	0%

P = n / N, where in the number of occurrences of a subclass (e.g., *PerformanceProbe* is a subclass of *Probe*), and N is the total number of occurrences of all sub-classes, e.g., *PerformanceProbe*, *EventProbe*, and *PhysicalProcessProbe* are all subclasses of *Probe*.

Appendix B Exectensions to Moka Execution Semantics

To execute ETRMs, we defined or extended the execution semantics for a few stereotypes from the *MoSH* profiles and UML metaclasses. They were implemented in the *TM-Executor* framework as extensions to Moka. This appendix gives the implementations of these newly defined execution semantics.

BroadcastSignalActionActivation

```
1.  // Construct a signal using the values from argument pins.
2.  // Send the signal to all objects that are associated with the
3.  // object from the source pin.
4.  doAction(){
5.    BroadcastSignalAction action = (BroadcastSignalAction) (this.node);
6.    Signal signal = action.getSignal();
7.    // instantiate signal
8.    SignalInstance signalInstance = new SignalInstance();
9.    signalInstance.type = signal;
10.   List<Property> attributes = signal.getOwnedAttributes();
11.   List<InputPin> argumentPins = action.getArguments();
12.   // set signal attributes
13.   for (int i = 0; i < attributes.size(); i++) {
14.     Property attribute = attributes.get(i);
15.     InputPin argumentPin = argumentPins.get(i);
16.     List<Value> values = this.takeTokens(argumentPin);
17.     signalInstance.setFeatureValue(attribute, values, 0);
18.   }
19.   Object_ object = (Object_)this.takeTokens(action.source).getValue(0);
20.   // broadcast signal
21.   for(FeatureValue featureValue : object.featureValues){
22.     Value value = featureValue.values.get(0);
23.     if(value instanceof Object_){
24.       ((Object_)value).send(signalInstance);
25.     }
26.   }
27. }
```

ChangeEventOccurrence

```
1.  // Evaluate registered change events.
2.  // If the change expression of a change event becomes true,
3.  // put the event in the object's event pool
4.  evaluateChangeEvent(){
5.    List<ChangeEvent> notTriggeredEvents = new ArrayList<ChangeEvent>();
6.    for(ChangeEvent event : this.registeredChangeEvents){
7.      // evaluate change expression of the change event
8.      if(event.evaluate()){
9.        // change expression becomes true,event occurs
10.       this.eventPool.add(event);
11.     }
12.     else{
13.       notTriggeredEvents.add(event);
14.     }
15.   }
16.   this.registeredChangeEvents = notTriggeredEvents;
17. }
```

InputOperationExecution

```
1.  // Take input parameter values as input.
2.  // Invoke the test interface corresponding to the URI defined in the
3.  // opaque behavior of the operation.
4.  execute(){
5.    OpaqueBehavior method = (OpaqueBehavior)this.methods.get(0);
```

```
6.    String uri = method.getBodies().get(0);
7.    InputInvocation invocation = new InputInvocation(uri);
8.    List<InputPin> inputPins = opaqueAction.getInputs();
9.    // set input
10.   for(InputPin inputPin : inputPins){
11.     String name = inputPin.getName();
12.     List<Value> values = this.takeTokens(inputPin);
13.     invocation.addInput(values);
14.   }
15.   // invoke test interface
16.   if(invocation.invoke() != 0){
17.     this.terminate();
18.   }
19. }
```

OutputOperationExecution

```
1.  // Invoke the test interface corresponding to the URI defined in the
2.  // opaque behavior of the operation.
3.  // Update attributes values using the outputs of the test interface.
4.  execute(){
5.    OpaqueBehavior method = (OpaqueBehavior)this.methods.get(0);
6.    String uri = method.getBodies().get(0);
7.    OutputInvocation invocation = new OutputInvocation(uri);
8.    // invoke test interface
9.    List<JSONObject> results = invocation.invoke();
10.   if(results == null){
11.     terminate();
12.   }
13.   // update attributes
14.   update( ((Object_)this.getExecutionContext()).featureValues, results );
15. }
```

StateActivation

```
1.  // Determine whether a state is enterable,
2.  // i.e., whether its state invariant is true.
3.  boolean isEnterable(){
4.    if(!this.getStateMachineExecution().getConfiguration().isActive()){
5.      return false; //the state machine is not active, cannot enter
6.    }
7.    if(this.stateInvariant == null) {
8.      return true;
9.    }
10.   while(true){
11.     if(invariant.evaluate()){
12.       break;     // state invariant becomes true, can enter now
13.     }
14.     else{
15.       if(timeout()){
16.         return false;     // timeout, stop waiting
17.       }
18.       ObjectActivation activation=getExecutionContext().objectActivation;
19.       synchronized(activation){
20.       try {
21.         activation.wait();       // wait the state invariant becomes true
22.       } catch (InterruptedException e) {
23.         e.printStackTrace();
24.       }
25.     }
26.   }
27.   return true;
28. }
```